#HashEx
BLOCKCHAIN SECURITY

Holygrails

smart contracts final audit report

December 2022


hashex.org


contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org). HashEx has exclusive rights to publish the results of this audit on company's web and social sites.

2. Overview

HashEx was commissioned by the **Holygrails** team to perform an audit of their smart contract. The audit was conducted between **2022-12-02** and **2022-12-14**.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available in the HolyGrails-Official repository. The audit was performed for commit 8c28798.

2.1 Summary

Project name	Holygrails
URL	https://www.holygrails.io/
Platform	Solana
Language	Rust

2.2 Contracts

Name	Address
airdrop	

sol_airdrop

staking

All programs

3. Found issues

Low	7 (78%)
Info	2 (22%)

C1. airdrop

ID	Severity	Title	Status
C1-01	Low	Lack of PDA checks	Resolved

C2. sol_airdrop

ID	Severity	Title	Status
C2-01	Low	Wrong size calculation	Resolved
C2-02	Low	Lack of PDA checks	Resolved

C3. staking

ID	Severity	Title	Status
C3-01	Low	Not full initialization	Resolved
C3-02	Low	PDA check	Resolved

C3-03	Low	Unused field	Resolved
C3-04	Info	The source of rewards	Resolved
C3-05	Info	Misleading code documentation	Resolved

C4. All programs

ID	Severity	Title	Status
C4-01	Low	Lack of custom error descriptions	Resolved

4. Contracts

C1. airdrop

Overview

The `airdrop` program is meant for creating and performing airdrops of SPL tokens.

A user can make an airdrop. Said user will become an admin of the airdrop, while other users can participate in this airdrop and create a data account. For tokens to be claimed, first the admin has to deposit them into the vault of the airdrop, and then a user has to ask the admin to sign his transaction for the claim.

Two signatures are needed to perform the claim: the user's and the admin's. Users can't claim tokens without the admin's signature. The amount of claimed tokens is determined by the airdrop's admin.

Issues

C1-01 Lack of PDA checks

Low Resolved

In all functions except `create_airdrop()`, the account `airdrop` isn't checked for being a PDA. All PDAs that are coming from a user should be checked. Checking ownership only isn't enough.

Recommendation

The structure `CreateReceiver` should be changed. For example, like this:

```
#[derive(Accounts)]
pub struct CreateReceiver<'info> {
    // Airdrop account
    #[account(
        mut,
```

```
        seeds = [
            b"AIRDROP".as_ref(),
            airdrop.creator.as_ref(),
            airdrop.token_mint.as_ref(),
        ],
        bump = airdrop.bump,
    )]
    pub airdrop: Account<'info, Airdrop>,

    ...
```

Add similar checks for `vault`, `receiver`, `airdrop` accounts in other functions.

C2. sol_airdrop

Overview

The `sol_airdrop` program is meant for creating and performing airdrops of native tokens.

A user can make an airdrop. Said user will become an admin of the airdrop, while other users can participate in this airdrop and create a data account. For tokens to be claimed, first the admin has to deposit them into the vault of the airdrop, and then a user has to ask the admin to sign his transaction for the claim.

Two signatures are needed to perform the claim: the user's and the admin's. Users can't claim tokens without the admin's signature. The amount of claimed tokens is determined by the airdrop's admin.

Issues

C2-01 Wrong size calculation

Low Resolved

The variable `Airdrop::SIZE` is larger than the actual size of the `Airdrop` struct by 32.

Recommendation

Lines 301-303 should be changed to this:

```
impl Airdrop {
    pub const SIZE: usize = 32 + 1 + (32 * 3) + (8 * 2);
}
```

C2-02 Lack of PDA checks

Low Resolved

In all functions except `create_airdrop()`, the account `airdrop` isn't checked for being a PDA. All PDAs that are coming from a user should be checked. Checking ownership only isn't enough.

Recommendation

The structure `CreateReceiver` should be changed. For example, like this:

```
#[derive(Accounts)]
pub struct CreateReceiver<'info> {
    // Airdrop account
    #[account(
        mut,
        seeds = [
            b"SOL_AIRDROP".as_ref(),
            airdrop.creator.as_ref(),
        ],
        bump = airdrop.bump,
    )]
    pub airdrop: Account<'info, Airdrop>,

    ...
```

Also, add similar checks for `receiver`, `airdrop` accounts in other functions.

C3. staking

Overview

The `staking` program is used for creating pools and staking in them.

Issues

C3-01 Not full initialization

Low Resolved

In the function `initialize_pool()`, there is no initialization of the `pending_admin` field in the structure `Pool`.

Recommendation

In the function `initialize_pool()`, add the following line:

```
pool.pending_admin = Pubkey::default();
```

C3-02 PDA check

Low Resolved

All PDAs that are coming from a user should be checked. Checking ownership only isn't enough.

Recommendation

The structure `CreateUser` should be changed. For example, like this:

```
#[derive(Accounts)]
pub struct CreateUser<'info> {
    // Stake instance.
    #[account(
        mut,
        constraint = !pool.paused @ ErrorCode::PoolPaused,
        seeds = [
```

```
            b"POOL",
            pool.creator.as_ref(),
        ],
        bump = pool.bump,
    )]
    pub pool: Box<Account<'info, Pool>>,

    ...
```

Also, add similar checks for `stake_token_vault`, `reward_token_vault`, `user` accounts in other functions.

C3-03 Unused field

Low Resolved

In the structure `StakeEntry` the field `stake_period` isn't used anywhere.

Recommendation

Reward calculation should be reconsidered or this field should be deleted.

Update

The team responded to the issue and added a comment in the code that this variable is used in UI to show the staker's inital intention for staking.

C3-04 The source of rewards

Info Resolved

The funds for the staking rewards are added externally. In order to function the program properly there thould be enough funds deposited.

Recommendation

Clarify the source of rewards in the project's documentation and make sure there are enough funds for the rewards.

Update

The team responded to the issue and added comment in the code that the project's admin or a third party are supposed to deposit enough funds for the stakers' rewards.

C3-05 Misleading code documentation

Info Resolved

In the structure `StakeEntry`, there are incorrect comments.

```
#[derive(AnchorSerialize, AnchorDeserialize, Eq, PartialEq, Copy, Clone)]
pub struct StakeEntry {
    /// Pool the this user belongs to.
    pub staked_amount: u64,
    /// The owner of this account.
    pub stake_period: u64,
    /// The staked amount by the user.
    pub staked_at: u64,
    /// The claimed amount by the user.
    pub withdrawn_at: u64,
    /// The number of stakes that the user have.
    pub claimed_amount: u64,
    /// The stake entries.
    pub active: bool,
}
```

Recommendation

Update the comments to conform with the code.

C4. All programs

Overview

Issues related to all programs in the scope of audit.

Issues

C4-01 Lack of custom error descriptions

Low Resolved

Custom error descriptions are added to the structs but not for all checks made in the structs.

For example:

```
#[derive(Accounts)]
#[instruction(amount: u64)]
pub struct Withdraw<'info> {
    /// The airdrop account.
    #[account(
        mut,
        has_one = admin @ ErrorCode::UnauthorizedAdmin,
        has_one = vault,
        has_one = token_mint
    )]
    pub airdrop: Account<'info, Airdrop>,
    ...
}
```

Recommendation

We recommend adding custom errors for every check to simplify debugging in case of errors.

Update

In the contract airdrop on lines 333,

5. Conclusion

No serious issues were found during the audit. The code is well-written and has unit tests for each program.

Appendix A. Issues severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Info.** Issues that do not impact the contract operation. Usually, info severity issues are related to code best practices, e.g. style guide.

Appendix B. Issue status description

Resolved. The issue has been completely fixed.

Partially fixed. Parts of the issue have been fixed but the issue is not completely resolved.

Acknowledged. The team has been notified of the issue, no action has been taken.

Open. The issue remains unresolved.

Appendix C. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

 contact@hashex.org

 @hashex_manager

 blog.hashex.org

 linkedin

 github

 twitter

#HashEx
BLOCKCHAIN SECURITY