Security Audit Report

Stakefish BatchDeposit Contract

Delivered: November 13th, 2020

Prepared for Stakefish by

Table of Contents

- Summary
- Disclaimer
- Minor Findings
  - Disable Inherited Function
  - Avoid Unnecessary Uses of Smaller Sized Integer Types
  - Avoid Unnecessary Uses of Division and Modulo Operation
  - Use Ether Units
  - Update Compiler Version
  - Front-Running
  - Business Logic Review
  - Bytecode-Level Test Coverage Analysis
- Common Anti-Pattern Analysis

Summary

Runtime Verification, Inc. conducted a security audit on the BatchDeposit contract written by the Stakefish team. Both the source code and the compiled bytecode were carefully reviewed, and *no critical issues* but a few minor issues were found. All the minor findings have been properly addressed in the latest version.

Scope

Below is the latest version of the contract that has been audited. It is ***important to double-check*** that the deployed version is identical to the following:[1]

- BatchDeposit.sol: source code of git-commit-id `a4912b2`
- BatchDeposit.bin (BatchDeposit.bin-runtime): bytecode compiled by the version 0.6.11 with the optimization enabled (`--optimize-runs 5000000`)

The audit is limited in scope within the boundary of the Solidity contract only. Off-chain and client-side portions of the codebase are *not* in the scope of this engagement. See our Disclaimer next.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have followed the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Second, we carefully checked if the code is vulnerable to known security issues and attack vectors. Third, we symbolically executed the bytecode of the contract to systematically search for unexpected, possibly exploitable, behaviors at the bytecode level, that are due to EVM quirks or Solidity compiler bugs. Finally, we employed Firefly to measure the test coverage at the bytecode level, identifying missing test scenarios, and helping to improve the quality of tests.

---

[1] For the bytecode, the last few bytes (starting from `0xa264`) refer to the metadata hash that can vary thus be ignored.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Minor Findings

Below are minor findings which have been properly addressed in the latest version.

Disable Inherited Function

The contract inherited OpenZeppelin's Ownable contract for access control, but disabled the `renounceOwnership()` function by overriding it with a "do-nothing" function (one with an empty body). It is a better practice for a deprecated function to revert rather than return silently.

Recommendation

Add revert in the body of the overriding function.

Status

Fixed in the latest version.

Avoid Unnecessary Uses of Smaller Sized Integer Types

The contract used `uint32` and `uint8` types, which was not necessary, rather could be problematic in case of overflow, and error-prone especially in potential code updates later. Uniformly using only `uint256` type is sufficient and cleaner for the current business logic, as well as slightly more gas-efficient by avoiding the truncation operations attached to arithmetics involving smaller typed values.

Recommendation

Refactor the code as suggested.

Status

Fixed in the latest version.

Avoid Unnecessary Uses of Division and Modulo Operation

The contract unnecessarily used div and mod operations, leading to poor code readability. A refactoring suggestion was made to avoid unnecessary uses of div and mod operations, as well as improve the code readability.

Recommendation

Refactor the code as suggested.

Status

Fixed in the latest version.

Use Ether Units

The contract introduced a constant GWEI to refer to 10^9, which could be avoided by using the builtin Ether units.

Recommendation

Replace GWEI with “1 gwei.”

Status

Fixed in the latest version.

Update Compiler Version

While the Solidity compiler version used for the Deposit contract deployed in the mainnet was 0.6.11, the BatchDeposit contract used 0.6.8. It was recommended to use the same compiler version to avoid any potential issues due to the version mismatch, unless there was a specific reason for sticking to the older version.

Recommendation

Use the Solidity compiler version 0.6.11, especially with the optimization enabled (`--optimize-runs 5000000`), as in the Deposit contract.

Status

Fixed in the latest version.

Front-Running

It is possible to front-run `batchDeposit()` transactions while a `changeFee()` transaction is pending.

Exploit Scenario

Suppose the contract owner submitted a `changeFee()` transaction to increase the fee. Seeing the transaction on the network, malicious users can front-run their `batchDeposit()` transactions (by submitting them with a higher gas price) to benefit from the current smaller fee schedule.

Recommendation

The above scenario can be prevented by changing the fee *only* when the contract is paused. That is, changing the fee will require executing `pause()`, followed by `changeFee()`, followed by `unpause()` functions. This restriction can be either internally incorporated into the contract operation policy, or externally implemented at the code level by explicitly adding the `whenPaused` modifier to the `changeFee()` function.

Business Logic Review

The following nontrivial aspects in the business logic of the BatchDeposit contract were identified.

While the Deposit contract allows us to deposit in installments (e.g., one can deposit 1 ETH as a trial, and then deposit the remaining 31 ETH later), the BatchDeposit contract doesn't allow that to avoid increasing the complexity of the frontend.

Multiple deposits processed by each `batchDeposit()` call are restricted to share the same `withdrawal_credentials` data.

Recommendation

Clarify these restrictions in the user document.

Bytecode-Level Test Coverage Analysis

The bytecode-level test coverage analysis powered by Firefly revealed missing test scenarios. For example, certain functions including inherited ones were never tested, and negative tests for certain `require()` assertions were missed.

Recommendation

Add more tests to cover missing cases.

Status

More tests were added. The latest coverage report is available here.

Common Anti-Pattern Analysis

We analyzed the safety of the contract against known security vulnerabilities. Below are the rationale of the safety against the vulnerabilities that are applicable to the contract.

Arithmetic Overflow

The contract adopted the SafeMath library for arithmetic operations whenever the absence of arithmetic overflow cannot be statically guaranteed at compile time.

Reentrancy

The contract involves two external contract calls. One is to call the `deposit()` function of the Deposit contract, which is trusted. Another is to transfer funds to a statically unknown address. For the latter, however, it utilizes the builtin `transfer()` function whose gas budget is restricted to only 2,300, which is small enough to prevent any potential reentrancy.[2]

Access Control

All of the functions designed for the contract owner are associated with the `onlyOwner` modifier. The visibility of all functions are explicitly specified.

Variable Shadowing

No variable names are clashed in the contract including the inherited ones.

Unexpected Ether

The contract logic does *not* depend on the current balance.

Dirty Higher Order Bits

The contract does *not* directly retrieve `msg.data`.

Unchecked External Calls

[2] The 2,300 gas stipend may not be safe against later hard-forks where the call gas goes down, but it would not be a major concern because the likelihood of such a hard-fork is small, considering many contracts already depend on it.

No low-level `call()` or `send()` functions are used in the contract.