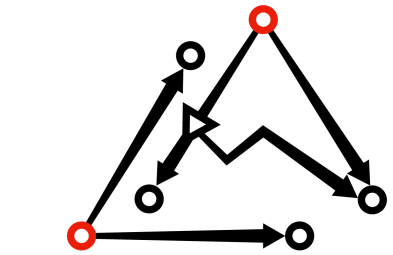

Least Authority

PRIVACY MATTERS

Smart Contracts

Security Audit Report

Fungify

Updated Final Audit Report: 10 January 2024

Table of Contents

Overview

Background

Fungify has requested that Least Authority perform a security audit of their smart contracts.

Project Dates

- **November 21, 2023 - December 13, 2023:** Initial Code Review *(Completed)*
- **December 15, 2023:** Delivery of Initial Audit Report *(Completed)*
- **January 10, 2024:** Verification Review *(Completed)*
- **January 10, 2024:** Delivery of Final Audit Report *(Completed)*
- **January 10, 2024:** Delivery of Updated Final Audit Report *(Completed)*

Review Team

- Nikos Iliakis, Security Researcher and Engineer
- Steven Jung, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Fungify smart contracts followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- Fungify Taki Contracts:
  https://github.com/fungify-dao/taki-contracts

Specifically, we examined the Git revision for our initial review:

- `a259825b8e1feea0f339d16e6565b6ba159017f2`

For the verification, we examined the Git revision:

- `2416d1724f90be9f7307d976ea5dd8c2ec859f72`

For the review, this repository was cloned for use during the audit and for reference in this report:

- Fungify Taki Contracts:
  https://github.com/LeastAuthority/fungify-taki-contracts-estimation

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Website:
  https://fungify.it

In addition, this audit report references the following documents:

- Blog post, “Introducing Fungify Lending Pools”: https://blog.fungify.it/p/introducing-fungify-lending-pools?utm_source=profile&utm_medium=reader2”
- Fungify | Protocol | Tools: https://docs.fungify.it/protocol/pools
- Solidity | Division: https://docs.soliditylang.org/en/v0.8.1/types.html?highlight=division#division

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the network;
- Potential misuse and gaming of the smart contracts;
- Attacks that impact funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) and other security exploits that would impact the intended use of the smart contracts or disrupt their execution;
- Vulnerabilities in the smart contracts’ code;
- Protection against malicious attacks and other ways to exploit the smart contracts;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team performed a security audit of Fungify's Pools smart contracts, which is a lending protocol extending Compound to facilitate NFT lending/borrowing with a unique interest market token.

In addition to auditing the security of the system and the areas of concern listed above, we examined the smart contracts for vulnerabilities and implementation errors and to assess adherence to best practice recommendations. More specifically, we scrutinized functions within the Markets vaults to identify potential logic errors, namely concerning liquidation and borrowing. We additionally reviewed the implementation for unchecked external smart contract calls, any kind of reentrancy attacks, front running attacks, storage collisions or unauthorized access, and incorrect mathematical operations, and could not identify any exploitable weaknesses.

System Design

Our team examined the design of the implementation and found that the Fungify smart contracts demonstrate robust security practices. The review revealed that security has been taken into consideration in the design and implementation of the contracts.

However, our team identified an issue whereby, at the initiation of a pool, an attacker can front run the first depositor, which can result in the depositor being unable to receive any tokens in exchange (Issue A).
In addition, we found that `doNFTTransferOut` always returns the last NFT in an array, which would allow a malicious user to target and retrieve a desired NFT from a pool (Issue B).

Code Quality

We performed a manual review of the repositories in scope and found that the Fungify smart contracts are generally well-organized and adhere to best practices. However, we identified several best practice suggestions that would improve overall quality, reliability, and readability of the codebase (Suggestion 4) (Suggestion 5) (Suggestion 6).

Tests

Our team found that sufficient test coverage of the smart contracts has been implemented.

Documentation and Code Comments

The documentation provided for this review offered an overview of the system, which was helpful. The diagrams included in the documentation are conceptual and give some technical details about the formulas used in the project.
Additionally, code comments sufficiently describe the intended behavior of security-critical components and follow the NatSpec format, thus facilitating the comprehension of the system.

Scope

The scope of this review included all security-critical components.

Dependencies

Our team examined all the dependencies implemented in the codebase and found that the implementation uses standard, well-audited libraries.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Minting Process Is Susceptible to Inflation / Sandwich Attacks	Resolved
Issue B: doNFTTransferOut Deterministically Returns the Last NFT	Resolved
Suggestion 1: Enter Market in receive Function	Resolved
Suggestion 2: Upgrade Solidity Version and Lock the Pragma	Resolved
Suggestion 3: Use Correct ERC20 Interface	Resolved
Suggestion 4: Initialize Variables Before Use	Resolved
Suggestion 5: Check Return Values of View Function Calls and collectInterest	Resolved
Suggestion 6: Avoid Comparison to Boolean Constants	Resolved

Issue A: Minting Process Is Susceptible to Inflation / Sandwich Attacks

Location

`contracts/CErc20.sol#L53`

`contracts/CToken.sol#L310`

`contracts/CErc20InterestMarket.sol#L154`

Synopsis

When a user mints CTokens, the `exchangeRateStoredInternal` function is eventually called, which checks the `totalSupply` of tokens in circulation. If the `totalSupply` is zero, the `exchangeRateStoredInternal` function uses the initial exchange rate; otherwise, it applies the formula: `exchangeRate = (totalCash + totalBorrows - totalReserves) / totalSupply`

At the initiation of a pool, if the pool is empty, an attacker can front run the first depositor and deposit a small amount. Consequently, due to the manner in which solidity handles rounding, the depositor could end up with zero CTokens.

Note that this attack is inherited by the fork of the Compound Protocol and is leveraged to exploit different projects.

Impact

This Issue could result in a depositor losing their shares and the attacker owning the `totalSupply`.

Preconditions

This attack can occur if the pool is empty at initiation, and an attacker has sufficient funds to execute the attack.

Technical Details

An attacker mints a small amount of CTokens (e.g. 100e4 wei) using the `mint` function and then transfers a large amount of underlying tokens (e.g.1e18 wei) directly to the CToken contract. As a result, the depositor mints, but obtains zero CTokens. The attacker would then own all the minted cTokens and have the ability to redeem them.

Remediation

We recommend preventing the pools from being empty. For example, one option could be to perform the first deposit in the same transaction as the deployment, thus preventing a malicious actor from front running it. However, given that such a remediation requires special attention and would need to be repeated multiple times, another solution would be to hard code a mint of some tokens to the zero address directly in the smart contract.

Status

The Fungify team acknowledged this Issue and decided to mint and burn at deployment to prevent it.

Verification

Resolved.

Issue B: doNFTTransferOut Deterministically Returns the Last NFT

Location

`contracts/CErc721.sol#L684`

Synopsis

Although users are aware that they will be unable to reclaim the exact NFT that they supplied, it is possible for a malicious user to inspect the pools and retrieve a desired NFT since `doNFTTransferOut` always returns the last NFT in the array.

Impact

An attacker could retrieve a desired NFT.

Preconditions

The attacker would need to have provided an NFT before the desired one was supplied.

Feasibility

Straightforward.

Remediation

We recommend that the Fungify team implement either the `Prevrandao opcode (fork >= paris)` or the Chainlink VRF to obtain a random number and retrieve a random NFT.

Status

The Fungify team acknowledged the Issue but stated that the Last-In First-Out (LIFO) method was by design and that users may receive different NFTs than the one they supplied to the pool.

Verification

Resolved.

Suggestions

Suggestion 1: Enter Market in receive Function

Location

`contracts/CEther.sol#L221`

Synopsis

The CEther contract implements a `receive` function, which is used to receive any Ether sent to the contract. The function calls the `mintInternal` function, which results in tokens being minted for `msg.sender`. However, it is never logged (stored) that the user uses this particular market.

The functions `autoEnterMarkets` (for users added by the protocol) and `enterMarket` (for users who add themselves) are used to include a user in the market. However, `autoEnterMarket` is not used in the `receive` function.

Although we could not find a way to exploit this, our team noted that it could lead to issues in further development in the future.

Mitigation

We recommend either calling `comptroller.autoEnterMarkets` before calling `mintInternal` or calling `mint` instead of `mintInternal`.

Status

The Fungify team has resolved this suggestion by calling `comptroller.autoEnterMarkets` before calling `mintInternal`.

Verification

Resolved.

Suggestion 2: Upgrade Solidity Version and Lock the Pragma

Location

All of the contracts.

Synopsis

A floating pragma is an error-prone practice that could lead to deployment issues in the case that an incorrect compiler version is used. The older compiler versions have known and fixed issues and can be used maliciously.

Mitigation

We recommend upgrading to the most recent compiler version, as it may include features and bug fixes for issues that were present in previous versions, and locking the pragma.

Status

The Fungify team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 3: Use Correct ERC20 Interface

Location

`contracts/CEther.sol#L7`

`contracts/CErc20.sol#L133`

Synopsis

The aforementioned contracts use an incorrect ERC20 interface. According to EIP-20, `approve`, `transfer`, and `transferFrom` should return Boolean values; however, some ERC20 tokens do not return the `result` value. The false `return` value could result in action failure and must therefore be taken into consideration when processing ERC20 tokens.

Mitigation

We recommend using the `SafeERC20` library to process ERC20 tokens.

Status

The Fungify team stated that they utilize `EIP20NonStandardInterface` where appropriate. Our team investigated the `EIP20NonStandardInterface` and did not identify any issues in the current implementation.

Verification

Resolved.

Suggestion 4: Initialize Variables Before Use

Location

`contracts/CToken.sol#L980`

Synopsis

The `actualAddAmount` function is not initialized before use.

Mitigation

We recommend explicitly setting a variable to zero to improve code readability.

Status

The Fungify team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 5: Check Return Values of View Function Calls and collectInterest

Location

`contracts/Comptroller.sol#L1151`

`contracts/CErc20.sol#L42`

`contracts/CErc721.sol#L43`

`contracts/CErc721.sol#L139`

`contracts/CErc721.sol#L253`

`contracts/CErc721.sol#L471`

Synopsis

In the aforementioned locations, the codes call `view` functions but do not use the `return` values. Additionally, the `collectInterest` function returns zero only if the transaction succeeds. In case of failure, other values are returned. Hence, no check is implemented to verify the `return` value of the function.

Mitigation

We recommend performing a sanity check on the `return` values of the functions.

Status

The Fungify team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 6: Avoid Comparison to Boolean Constants

Location

contracts/Comptroller.sol#L156

contracts/Comptroller.sol#L1248

contracts/Comptroller.sol#L1264

contracts/Comptroller.sol#L1277

contracts/Comptroller.sol#L1290

contracts/Comptroller.sol#L1303

contracts/Comptroller.sol#L1316

contracts/Comptroller.sol#L1383

Synopsis

Boolean variables can be used directly as an `if` condition and do not need to be compared against `true` or `false`.

Mitigation

We recommend using Boolean variables directly without comparing them against `true` or `false`.

Status

The Fungify team has implemented the mitigation as suggested.

Verification

Resolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.