hexens × Fungify.

Nov.23

# SECURITY REVIEW REPORT FOR FUNGIFY

# CONTENTS

# ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

# AUDIT LED BY

## KASPER ZWIJSEN

Head of Smart Contract Audits | Hexens

---

Audit Starting Date
23.11.2023

Audit Completion Date
04.12.2023

---

hexens × Fungify.

# METHODOLOGY

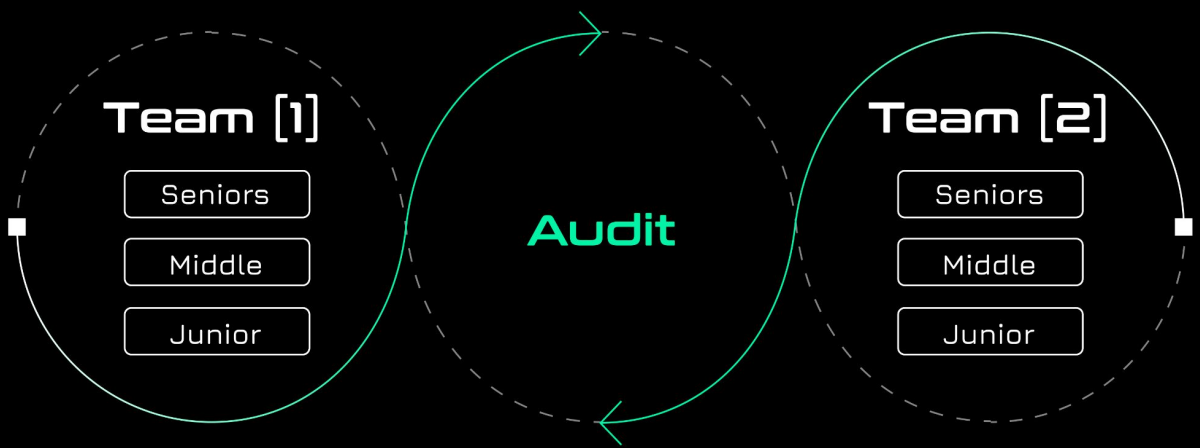
## COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

Auditor* Audit

## HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



# SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

| IMPACT | PROBABILITY | | | |
|---|---|---|---|---|
| | Rare | Unlikely | Likely | Very Likely |
| Low / Info | Low / Info | Low / Info | Medium | Medium |
| Medium | Low / Info | Medium | Medium | High |
| High | Medium | Medium | High | Critical |
| Critical | Medium | High | Critical | Critical |

# SEVERITY CHARACTERISTICS

Vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of vulnerabilities:

## CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

## HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

## MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

## LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

## INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of the project. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

# EXECUTIVE SUMMARY

## OVERVIEW

This audit covered the "Pools" contracts of Fungify, a new lending protocol that builds on Compound to support lending/borrow of NFTs and introduces a special interest market token that is linked to NFT markets.

Our security assessment was a second review of the smart contracts, after completion of our first engagement.

During our audit, we have identified 1 critical severity vulnerability. It would allow the interest markets to be drained of all of its underlying tokens.

We have also identified 1 medium severity vulnerability.

Finally, all of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

# SCOPE

The analyzed resources are located on:
https://github.com/fungify-dao/taki-contracts/commit/b13d6f10c09d5594e0ee41ac63ed8516f05bab52

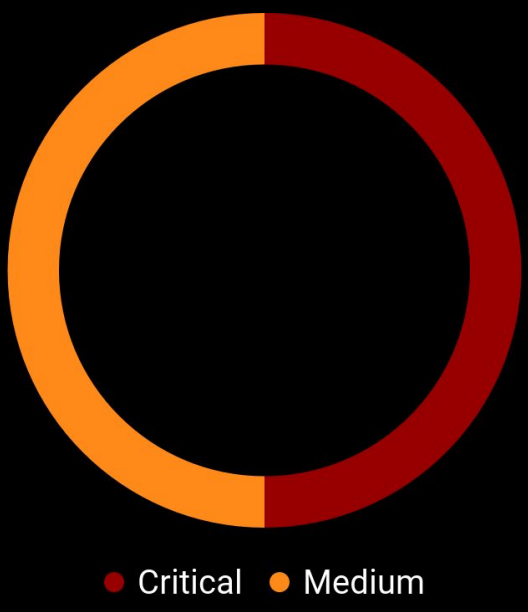
The issues described in this report were fixed in the following commit:
https://github.com/fungify-dao/taki-contracts/commit/458b331153d5d04ea90bce85c3fcb19e80c9b598

# SUMMARY

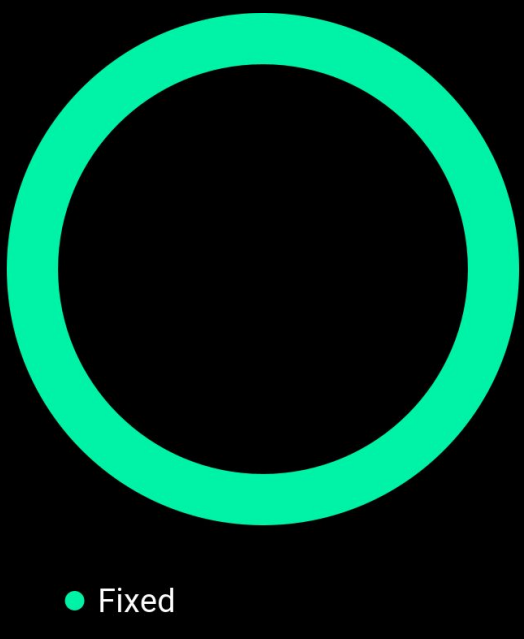
| SEVERITY | NUMBER OF FINDINGS |
| --- | --- |
| CRITICAL | 1 |
| HIGH | 0 |
| MEDIUM | 1 |
| LOW | 0 |
| INFORMATIONAL | 0 |

**TOTAL: 2**

## SEVERITY




## STATUS



# WEAKNESSES

This section contains the list of discovered weaknesses.

## FNG-20. ASSETS FROM INTEREST MARKETS CAN BE STOLEN THROUGH REWARD MANIPULATION

SEVERITY: Critical

PATH: CErc721.sol:_seize:L544-578

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The updating of the token balance of a user should be accompanied by the updating of the supply interest index and accrued of the user.

If not, the interest rewards to-be-claimed can be subject to manipulation by increasing the balance before calculation, resulting in a greater amount of rewards depending on the balance and time span.

Similar to "**FNG-11: All interest tokens can be stolen from the interest market**", this issue still exists in the **CErc721.sol:_seize** where the collateral of the borrower is transferred to the liquidator through directly updating the **accountTokens** mapping. The function does not update the **supplyInterest** (which is now only done upon normal transfers).

As a result, the token balance of the liquidator increases before the interest rewards calculation and so the new balance will also be counted over the differences in index and time compared to the last time it was updated.

This can be exploited by having some built up interest from the index and enough accounts (e.g. using an exploit contract) to completely drain the interest market from all its tokens, as it can be used to fabricate rewards instantly and those can be collected from the interest market in its underlying tokens.

```
function _seize(address liquidator, address borrower, uint seizeTokens) override external nonReentrant returns (uint) {
    if (msg.sender != address(comptroller)) {
        revert Unauthorized();
    }

    accrueInterest();

    uint oneNFTAmount = doubleScale / exchangeRateStoredInternal();
    if (seizeTokens % oneNFTAmount != 0) {
        // ensure whole nft seize size by rounding up to the next whole NFT
        seizeTokens = ((seizeTokens / oneNFTAmount) + 1) * oneNFTAmount;
    }

    /* Fail if borrower = liquidator */
    if (borrower == liquidator) {
        revert LiquidateSeizeLiquidatorIsBorrower();
    }

    uint liquidatorSeizeTokens = seizeTokens;

    ////////////////////////
    // EFFECTS & INTERACTIONS
    // (No safe failures beyond this point)

    /* We write the calculated values into storage */
    accountTokens[borrower] = accountTokens[borrower] - seizeTokens;
    accountTokens[liquidator] = accountTokens[liquidator] + liquidatorSeizeTokens;

    /* Emit a Transfer event */
    emit Transfer(borrower, liquidator, liquidatorSeizeTokens);
    //emit Transfer(borrower, address(this), protocolSeizeTokens);
    //emit ReservesAdded(address(this), protocolSeizeAmount, totalReservesNew);

    return seizeTokens;
}
```

The remediation is similar to “**FNG-11: All interest tokens can be stolen from the interest market**”. The **supplyInterest** mapping should be updated for both users, before moving balances.

For example:

```
function _seize(address liquidator, address borrower, uint seizeTokens) override external nonReentrant returns (uint) {
    if (msg.sender != address(comptroller)) {
        revert Unauthorized();
    }

    accrueInterest();
>>>>    supplyInterest[borrower].interestAccrued = supplyInterestStoredInternal(src);
>>>>    supplyInterest[borrower].interestIndex = supplyIndex;
>>>>    supplyInterest[liquidator].interestAccrued = supplyInterestStoredInternal(dst);
>>>>    supplyInterest[liquidator].interestIndex = supplyIndex;

    uint oneNFTAmount = doubleScale / exchangeRateStoredInternal();
    if (seizeTokens % oneNFTAmount != 0) {
        // ensure whole nft seize size by rounding up to the next whole NFT
        seizeTokens = ((seizeTokens / oneNFTAmount) + 1) * oneNFTAmount;
    }

    /* Fail if borrower = liquidator */
    if (borrower == liquidator) {
        revert LiquidateSeizeLiquidatorIsBorrower();
    }

    uint liquidatorSeizeTokens = seizeTokens;

    ////////////////////////
    // EFFECTS & INTERACTIONS
    // (No safe failures beyond this point)

    /* We write the calculated values into storage */
    accountTokens[borrower] = accountTokens[borrower] - seizeTokens;
    accountTokens[liquidator] = accountTokens[liquidator] + liquidatorSeizeTokens;

    /* Emit a Transfer event */
    emit Transfer(borrower, liquidator, liquidatorSeizeTokens);
    //emit Transfer(borrower, address(this), protocolSeizeTokens);
    //emit ReservesAdded(address(this), protocolSeizeAmount, totalReservesNew);

    return seizeTokens;
}
```

# FNG-9. MISSING SLIPPAGE PROTECTION IN THE LIQUIDATEBORROW FUNCTION

SEVERITY: Medium

PATH: Comptroller.sol:L294-379

REMEDIATION: add e.g. minimumLiquidatedValue parameter and add a check verifying that the reward value is greater than or equal to the minimumLiquidatedValue value

STATUS: fixed

DESCRIPTION:

The **batchLiquidateBorrow** function in **Comptroller.sol** calculates the amount of funds that will be transferred in a liquidation. This computation relies on asset prices retrieved from an oracle through the **oracle_.getUnderlyingPrice call**. However, it's important to highlight the absence of slippage protection in this process.

There's no guarantee that the amount returned by the **batchLiquidateBorrow** function corresponds to the current market price. This is because the transaction that updates the price feed might be mined before the actual call to batchLiquidateBorrow. As a result, users may experience liquidation amounts that differ from their expectations.

Consider a scenario where a user initiates a batchLiquidateBorrow function. Due to an oracle malfunction or significant price fluctuations, the amount of collateral transferred from the module might be much lower than the user would have anticipated based on the current market conditions.

It is recommended to introduce a parameter that would empower the caller to set a threshold for the minimum acceptable value for the liquidation. By doing so, users can explicitly state their assumptions about the liquidation and ensure that the collateral payout remains as profitable as expected.

```
  function batchLiquidateBorrow(address borrower, Liquidatables[] memory liquidatables, CTokenInterface[] memory
cTokenCollaterals) external nonReentrant returns (uint[][2] memory results) {
    require(adminWhitelist[msg.sender], "unauthorized");
    require(!seizeGuardianPaused, "seize is paused");

    (uint err, uint beforeRatio) = getAccountDebtRatio(borrower);
    require(err == uint(Error.NO_ERROR) && beforeRatio != 0, "INSUFFICIENT_SHORTFALL");

    results[0] = new uint[](liquidatables.length);
    results[1] = new uint[](cTokenCollaterals.length);

    PriceOracle oracle_ = oracle;

    uint liquidatedValueTotal;
    {
      for (uint i = 0; i < liquidatables.length;) {

        require(markets[liquidatables[i].cToken].isListed, "market not listed");

        if (liquidatables[i].amount != 0) {
          results[0][i] = CErc20Interface(liquidatables[i].cToken)._liquidateBorrow(msg.sender, borrower, liquidatables[i].amount);
        } else {
          results[0][i] = CErc721Interface(liquidatables[i].cToken)._liquidateBorrow(msg.sender, borrower, liquidatables[i].nftIds);
        }

        uint priceMantissa = oracle_.getUnderlyingPrice(CToken(liquidatables[i].cToken));
        require(priceMantissa != 0, "PRICE_ERROR");

        liquidatedValueTotal = mul_ScalarTruncateAddUInt(Exp({mantissa: priceMantissa}), results[0][i], liquidatedValueTotal);

        unchecked { i++; }
      }
      require(liquidatedValueTotal != 0, "error");
    }
```

```
{
    uint liquidatedValueRemaining = liquidatedValueTotal;
    for (uint i = 0; i < cTokenCollaterals.length && liquidatedValueRemaining != 0;) {

        require(markets[address(cTokenCollaterals[i])].isListed, "market not listed");

        {
            /* We calculate the number of collateral tokens that will be seized */
            uint seizeTokens = liquidateCalculateSeizeTokensNormed(address(cTokenCollaterals[i]), liquidatedValueRemaining);
            uint actualSeizeTokens;

            uint borrowerBalance = cTokenCollaterals[i].balanceOf(borrower);
            if (borrowerBalance < seizeTokens) {
                // can't seize more collateral than owned by the borrower
                actualSeizeTokens = borrowerBalance;
            } else {
                actualSeizeTokens = seizeTokens;
            }

            actualSeizeTokens = cTokenCollaterals[i]._seize(msg.sender, borrower, actualSeizeTokens);
            require(actualSeizeTokens != 0, "token seizure failed");

            uint actualRepayAmount = liquidatedValueRemaining;
            if (actualSeizeTokens != seizeTokens) {
                actualRepayAmount = actualRepayAmount * actualSeizeTokens / seizeTokens;
            }

            liquidatedValueRemaining = liquidatedValueRemaining > actualRepayAmount ?
                liquidatedValueRemaining - actualRepayAmount :
                0;

            results[1][i] = actualSeizeTokens;
        }

        unchecked { i++; }
    }

    require(liquidatedValueRemaining == 0, "LIQUIDATE_SEIZE_TOO_LITTLE");
}
```

```
    /* We emit a LiquidateBorrow event */
    //emit LiquidateBorrow(liquidator, borrower, nftIds, repayInterest, cTokenCollaterals, seizeTokensList);

    uint afterRatio;
    (err, afterRatio) = getAccountDebtRatio(borrower);

    // we allow Error.TOO_LITTLE_INTEREST_RESERVE here as long as debt ratio is improved or unchanged
    require((err == uint(Error.NO_ERROR) || err == uint(Error.TOO_LITTLE_INTEREST_RESERVE))
        && afterRatio <= beforeRatio, "seize too much");

    return results;
}
```