hexens × Fungify.

Oct.23

# SECURITY REVIEW REPORT FOR FUNGIFY

# CONTENTS

# 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
16.10.2023

Audit Completion Date
06.11.2023

---

hexens × Fungify.

# METHODOLOGY

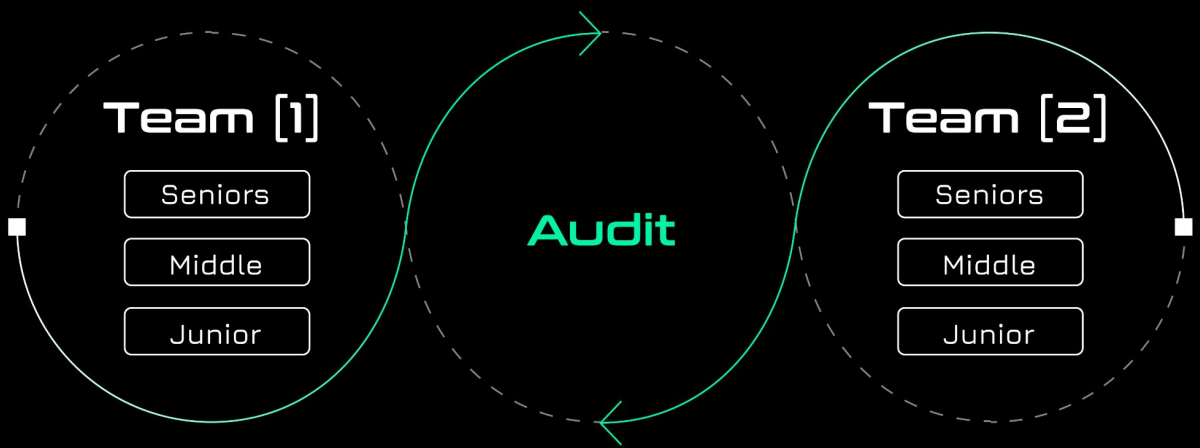
## COMMON AUDIT PROCESS

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 full review of the smart contracts.

During our audit, we have identified 2 critical severity vulnerabilities. The first one would allow the interest markets to be drained of all of its underlying tokens. The second one would allow bypassing a health check and escaping collateral to create collateral-free loans and bad debt for the protocol.

We have also identified 2 high severity vulnerabilities, various minor vulnerabilities and code optimisations.

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/b633740935c8b45a33833f753979afd06acea3f3

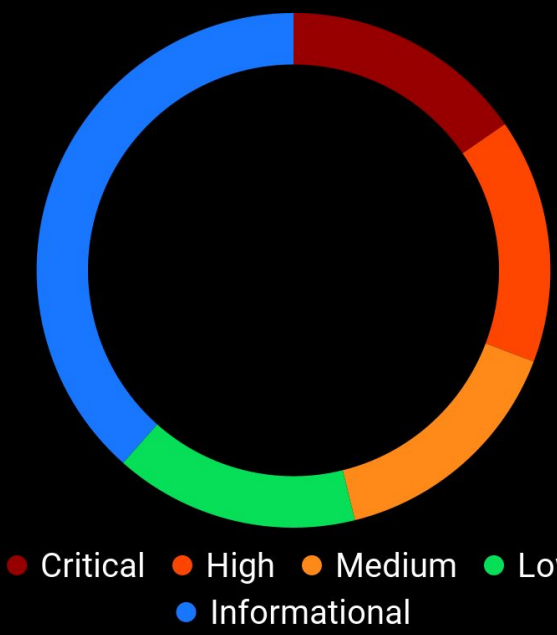
The issues described in this report were fixed in the following commit:
https://github.com/fungify-dao/taki-contracts/commit/271f22fbf32c760ce68c53c877dfb2d6458232ae

# SUMMARY

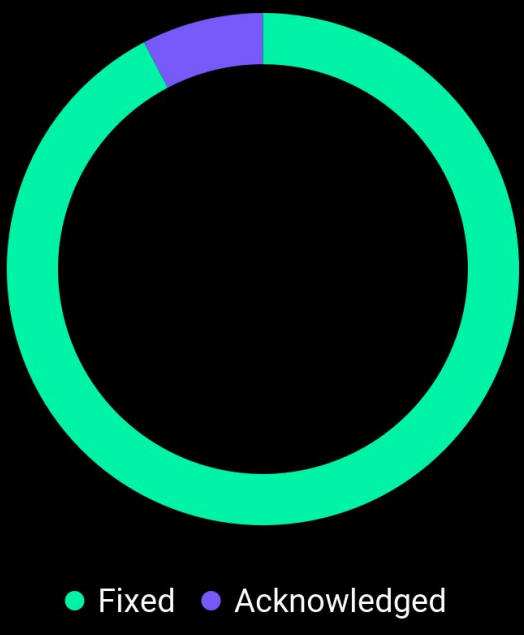
| SEVERITY | NUMBER OF FINDINGS |
|---|---|
| CRITICAL | 2 |
| HIGH | 2 |
| MEDIUM | 2 |
| LOW | 2 |
| INFORMATIONAL | 5 |

**TOTAL: 13**

## SEVERITY




## STATUS



# WEAKNESSES

This section contains the list of discovered weaknesses.

## FNG-11. ALL INTEREST TOKENS CAN BE STOLEN FROM THE INTEREST MARKET

SEVERITY: Critical

PATH: CErc721.sol:supplyInterestStoredInternal:L752-763

REMEDIATION: the CErc721 contract should override the _beforeTokenTransfer hook and force a claiming of interest rewards for both the sender and receiver

STATUS: fixed

DESCRIPTION:

The NFT underlying CToken, CErc721, uses a CErc20InterestMarket as tokens for both interest rewards for suppliers and interest payments for borrowers. In order to calculate the rewards for those supplies, it uses a supply index and **interestIndex** that is stored in the user's snapshot such that past rewards aren't counted (e.g. after claiming).

The difference between the new **supplyIndex** and **interestIndex** is then multiplied with the token balance of the user to calculate a supplier's earned interest.

However, this process does not take transfers into account and this can be used to create arbitrary interest rewards. More specifically, when the shares (CTokens) are transferred to another user, the balance increases,

but it does not set the **interestIndex** for the receiving user (e.g. by forcing a claim in a **_beforeTokenTransfer** hook).

In other words, a balance increase occurs and thus the receiving user can claim rewards over the new balance without actually having held this balance over the reward period.

For example:

1. User A supplies 1 NFT and mints 100 CTokens.
2. User B-Z call a reward claim to set their **interestIndex** (otherwise a division by 0 would occur).
3. After some time, the **supplyIndex** has increased.
4. User A can now claim rewards and transfer the balance of 100 CTokens to user B.
5. User B can now claim the same rewards over the new balance and then transfer the balance to user C.
6. Repeat step 5 for all users.

By having some built up interest from the index and enough accounts (e.g. using an exploit contract) an attacker could instantly and completely drain the interest market of all its underlying tokens.

```
function supplyInterestStoredInternal(address account) internal view returns (uint) {
    Exp memory exchangeRate = Exp({mantissa: exchangeRateStored()});
    uint supplyBalance = mul_ScalarTruncate(exchangeRate, accountTokens[account]);

    if (supplyBalance == 0) {
        return 0;
    }

    SupplyInterestSnapshot storage supplyInterestSnapshot = supplyInterest[account];
    uint newInterestAccrued = (supplyBalance * supplyIndex / supplyInterestSnapshot.interestIndex) -
supplyBalance;
    return supplyInterestSnapshot.interestAccrued + newInterestAccrued;
}
```

## FNG-17. CERC20INTERESTMARKET COLLATERAL CAN BE DIRECTLY USED TO PAY INTEREST AND BYPASS HEALTH CHECK

SEVERITY: Critical

PATH: CErc20InterestMarket.sol:payInterestInternal:L67-101

REMEDIATION: the function payInterestInternal should do a check by calling to Comptroller.isRedeemAllowed for the payer and the payment amount in a require statement. Similar to the redeem function

STATUS: fixed

DESCRIPTION:

A CErc20InterestMarket is a CERC20 token that can be used to pay interest for a CERC721 loan. These tokens have USD stable coins as underlying, such as USDC. These tokens are also normal CERC20 tokens and can be used as collateral for loans, so any balance change should be accompanied by a health check.

However, the function to pay interest for a CERC721 loan directly takes from the balance of the payer using **payInterestInternal**. If that balance was used as collateral in a loan, then this would bypass the health check completely, making the loan insolvent.

This provides a way to instantly (in a single transaction) create loans without any collateral backing it, creating large bad debt for the protocol.

For example:

1. A user has a loan for an ERC721 worth 100 ETH.
2. After some time, the user racks up 10 ETH worth in interest for that loan.
3. The user now uses a second account to create a new loan using 10 ETH worth of USDC as collateral and borrowing as much as possible (e.g. 8.5 ETH worth of DAI).
4. The user now calls **CErc721:repayBorrowBehalf**, which will call **CErc20InterestMarket:payInterestInternal** and pay the interest using the collateral balance of the second account, bypassing the health check.
5. In the end, only 1.5 ETH was paid back instead of the 10 ETH and an 8.5 ETH bad debt loan has been created.

```
function payInterestInternal(address borrowMarket, address payer, uint interestAmount) internal {
    if (interestAmount == 0) {
        return;
    }

    /* Fail if pay interest not allowed */
    uint allowed = comptroller.payInterestAllowed(address(this), borrowMarket, payer, interestAmount);
    require(allowed == 0, "pay not allowed");

    Exp memory exchangeRate = Exp({mantissa: exchangeRateStoredInternal()});
    uint interestTokens = div_(interestAmount, exchangeRate);

    // payer interest market balance is reduced to cover interest being paid
    uint balancePayer = accountTokens[payer];
    require(balancePayer >= interestTokens, "insufficient payer reserve");
    accountTokens[payer] = balancePayer - interestTokens;
    emit Transfer(payer, address(0), interestTokens);

    uint totalVirtual_ = totalVirtual;
    uint heldBalance;
    if (interestTokens > totalVirtual_) {
        heldBalance = interestTokens - totalVirtual_;
        totalSupply = totalSupply - totalVirtual_;
        totalVirtual = 0;
    } else {
        totalSupply = totalSupply - interestTokens;
        totalVirtual = totalVirtual_ - interestTokens;
    }

    if (heldBalance != 0) {
        // keep a reserve of cToken
        accountTokens[address(this)] = accountTokens[address(this)] + heldBalance;
        emit Transfer(address(0), address(this), heldBalance);
    }
}
```

# FNG-15. CERC721 LIQUIDATION WILL ALWAYS REVERT IF INTEREST STATE IS UP-TO-DATE

SEVERITY: High

PATH: CErc721.sol:_liquidateBorrow:L428-464

REMEDIATION: either accrueInterest() should return the exchange rate in the short-circuit branch, or _liquidateBorrow should handle this case correctly

STATUS: fixed

DESCRIPTION:

The function **_liquidateBorrow** of the CErc721 contract fetches the current exchange rate using accrueInterest() on line 431:

```
uint assetsExchangeRate = accrueInterest();
```

Afterwards, this value is used in a check for freshness on lines 436-438:

```
if (accrualBlockNumber != getBlockNumber() || assetsExchangeRate == 0) {
    revert LiquidateFreshnessCheck();
}
```

However, the function **accrueInterest()** only returns the exchange rate at the end of the function after the interest state has been updated. The function will short-circuit if the interest state had already been updated in the same block (i.e. **accrualBlockNumber** is equal to **block.number**), which can be seen in lines 651-653:

```
if (accrualBlockNumberPrior == currentBlockNumber) {
    return NO_ERROR;
}
```

In this case, it will return **NO_ERROR** (**0**) and consequently the **_liquidateBorrow** function will always revert due to the freshness check.

As a result, liquidations will always fail if there were any interactions with the CToken, if there are multiple liquidations in one block and liquidations could be blocked by front-running with a call to **accrueInterest()**.

```
function _liquidateBorrow(address liquidator, address borrower, uint[] memory nftIds) override external nonReentrant
returns (uint) {
    require(msg.sender == address(comptroller), "unauthorized");

    uint assetsExchangeRate = accrueInterest();

    uint repayAmount = nftIds.length * expScale;

    /* Verify market's block number equals current block number */
    if (accrualBlockNumber != getBlockNumber() || assetsExchangeRate == 0) {
        revert LiquidateFreshnessCheck();
    }

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

    /* Fail if repayAmount = 0 */
    if (repayAmount == 0) {
        revert LiquidateCloseAmountIsZero();
    }

    /* Fail if repayBorrow fails */
    uint repayInterest;
    (repayAmount, repayInterest) = repayBorrowFresh(liquidator, borrower, nftIds, 0);

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

    // convert interest value to NFT units
    repayInterest = repayInterest * expScale / assetsExchangeRate;

    // combine in NFT unit terms for seizure calculation
    uint actualRepayAmount = repayAmount + repayInterest;

    return actualRepayAmount;
}
```

## FNG-16. LOANS WITH CERC721 COLLATERAL CAN BE MADE UNLIQUIDATABLE

SEVERITY: High

PATH: CErc721.sol:_seize:L470-508

REMEDIATION: the combination of transferrable CErc721 shares and the rounding for full NFTs are a source of trouble. We would like to recommend to not round up to the nearest NFT on liquidation

we do not recommend making CErc721 shares non-transferrable. Even though this would mitigate the issue, it highly impacts the user experience and goes against the essence of the protocol. It would also not allow any 3rd party protocol to integrate with CErc721 (e.g. DEXs or yield aggregators)

STATUS: fixed

DESCRIPTION:

The **_seize** function for CErc721 will always round up to a token amount for full NFTs. This function is used for reward a liquidator in **Comptroller.sol:batchLiquidateBorrow** with the value that came from a liquidated debt of a borrower.

If the liquidation value is less than a full NFT amount, then the amount is rounded up to the nearest NFT on lines 475-479:

```
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;
}
```

However, this does work if the borrower owns less than 1 NFT in collateral, e.g. by transferring come CNFT tokens to another address and then creating a loan.

As a result, any loan that has a CErc721 as collateral can be forcefully made unliquidatable by the borrower, as the liquidation process would always revert.

For example:

1. Borrower mints 1 CNFT from their NFT.
2. Borrower transfer 1 wei CNFT to another address, keeping 0.9999 CNFT shares.
3. Borrower can now create a loan with the 0.9999 CNFT, one that is almost as large as with the full NFT, e.g. 0.84999 the value of the NFT in USDC (with an 85% collateral factor).
4. The liquidator can now never liquidate this loan, as it would result in an underflow revert, since the borrower does not own 1 CNFT.

This will create a strategy for borrowers to always profit from the protocol.

For example, if the loan become unhealthy, the protocol cannot liquidate this to obtain the collateral, creating bad debt. The borrower would be protected from the drop in price of their collateral. On the other hand, if the collateral increases in price, the borrower can simply repay the loan and re-obtain their collateral.

```
function _seize(address liquidator, address borrower, uint seizeTokens) override external nonReentrant returns (uint) {
    require(msg.sender == address(comptroller), "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 seize not allowed */
    /*uint allowed = comptroller.seizeAllowed(address(this), seizerToken, liquidator, borrower, seizeTokens);
    if (allowed != 0) {
        revert LiquidateSeizeComptrollerRejection(allowed);
    }*/

    /* 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-2. INADEQUATE CONSTRAINTS ON SEIZE SHARE MANTISSA IN THE PROTOCOL

SEVERITY: Medium

PATH: CToken.sol:L878-894

REMEDIATION: define an upper bound or a reasonable range for the protocolSeizeShareMantissa parameter to prevent it from being set too high or too low

STATUS: fixed

DESCRIPTION:

In the current implementation of the protocol, the **protocolSeizeShareMantissa** parameter in comparison with Compound is not constant (**CTokenInterface.sol** line 113) and can be adjusted by the admin. This lack of constraints could potentially introduce risks and imbalances in the protocol's operation.Allowing the admin to modify this parameter without appropriate restrictions might lead to the following concerns:

1. **Risk of Overcapitalization:** If an admin decides to set the **protocolSeizeShareMantissa** too high, it could lead to overcapitalization of the reserves, potentially causing inefficiencies in the allocation of assets and negatively impacting users' earnings.

2. **Imbalance in Collateral Seizure:** An excessively high **protocolSeizeShareMantissa** may incentivize users to seek liquidations, potentially leading to a sudden surge in collateral seized by the protocol. This could create an imbalance in the ecosystem and undermine stability.

```
function _setProtocolSeizeShare(uint newProtocolSeizeShareMantissa) virtual override external returns (uint) {
    // Check caller is admin
    if (msg.sender != admin) {
        revert SetReserveFactorAdminCheck();
    }

    // Save current value for use in log
    uint oldProtocolSeizeShareMantissa = protocolSeizeShareMantissa;

    // Set liquidation incentive to new incentive
    protocolSeizeShareMantissa = newProtocolSeizeShareMantissa;

    // Emit event with old incentive, new incentive
    emit NewProtocolSeizeShare(oldProtocolSeizeShareMantissa, newProtocolSeizeShareMantissa);

    return NO_ERROR;
}
```

## FNG-8. ALL NFTS ARE EVALUATED AT THEIR FLOOR PRICE AND CAN LEAD TO USER'S LOSS

SEVERITY: Medium

PATH: CErc721.sol:doNFTTransferOut:L624-637

REMEDIATION: this constitutes to a user risk and users holding NFTs with a significant higher price than the floor price should at least be warned by the front-end that their NFT won't be evaluated at its price

STATUS: acknowledged, see commentary

DESCRIPTION:

In the **CErc721.sol** when the user mints CTokens with the **mint()** function the user supplies **nftIds** which they would like to exchange for CTokens. But the **exchangeRate** is fixed for any NFT in that collection. This might lead to NFTs with different prices leading to the user getting the same amount of CTokens, as all NFTs are basically evaluated at their floor price.

The same price calculation is missing in the **redeem()** function which once again leads to that same problem. For example the Azuki NFTs, which according to the protocols documentation is supported, currently has NFT **Azuki #1725** which was sold for 7.30 ETH and **Azuki #5544** which was sold for 4.135 ETH. In case both of those users decided to mint their NFTs in the **CErc721.sol** and while their prices have significant difference both of them would get the same amount of CTokens.

This can lead to malicious user changing his low value NFT for a much higher value NFT because of the way the **doNFTTransferOut()** is implemented.

The documentation claims that when the user calls the r**edeem()** function the protocol would use Chainlink's VRF to get a verifiable random number to give a random NFT to the user who called the **redeem()** function, but instead the protocol gives the last NFT that was deposited into the protocol to the user. This can be seen in the implementation of the **doNFTTransferOut()** where the protocol transfers the last element of the heldNFTs array which contains the ids of all of the NFTs that were deposited into the contract. Thus the following case might happen:

- The malicious user deposits a cheap NFT for some x amount of tokens in the **CErc721.sol**
- A normal user comes and deposits his expensive NFT into the same contract
- The malicious user seeing that a more expensive NFT was deposited into the contract and because all of the NFTs have a flat **exchangeRate** with CTokens, the malicious user redeems his tokens for the more expensive NFT thus changing his cheap NFT for a more expensive NFT

```
function mint(uint[] memory nftIds) external override nonReentrant returns (uint) {

    comptroller.autoEnterMarkets(msg.sender); // silent failure allowed

    accrueInterest();

    uint mintAmount = nftIds.length * expScale;
    address minter = msg.sender;

    /* Fail if mint not allowed */
    uint allowed = comptroller.mintAllowed(address(this), minter, mintAmount);
    if (allowed != 0) {
        revert MintComptrollerRejection(allowed);
    }

    /* Verify market's block number equals current block number */
    if (accrualBlockNumber != getBlockNumber()) {
        revert MintFreshnessCheck();
    }

    Exp memory exchangeRate = Exp({mantissa: exchangeRateStoredInternal()});

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

    supplyInterest[minter].interestAccrued = supplyInterestStoredInternal(minter);
    supplyInterest[minter].interestIndex = supplyIndex;

    uint actualMintAmount = doNFTTransferIn(minter, nftIds) * expScale;

    /*
     * We get the current exchange rate and calculate the number of cTokens to be minted:
     *  mintTokens = actualMintAmount / exchangeRate
     */

    uint mintTokens = div_(actualMintAmount, exchangeRate);
```

```
    /*
     * We calculate the new total supply of cTokens and minter token balance, checking for overflow:
     *  totalSupplyNew = totalSupply + mintTokens
     *  accountTokensNew = accountTokens[minter] + mintTokens
     * And write them into storage
     */
    totalSupply = totalSupply + mintTokens;
    accountTokens[minter] = accountTokens[minter] + mintTokens;

    /* We emit a Mint event, and a Transfer event */
    emit Mint(minter, actualMintAmount, mintTokens, nftIds);
    emit Transfer(address(this), minter, mintTokens);

    /* We call the defense hook */
    // unused function
    // comptroller.mintVerify(address(this), minter, actualMintAmount, mintTokens);

    return NO_ERROR;
}
```

```
function redeemFresh(address payable redeemer, uint redeemTokensIn, uint redeemAmountIn) internal override {
    require(redeemTokensIn == 0 || redeemAmountIn == 0, "one of redeemTokensIn or redeemAmountIn must be zero");

    // NFT count -> amount
    redeemAmountIn = redeemAmountIn * expScale; //1e18

    /* exchangeRate = invoke Exchange Rate Stored() */
    Exp memory exchangeRate = Exp({mantissa: exchangeRateStoredInternal() });

    uint redeemTokens;
    uint redeemAmount;
    /* If redeemTokensIn > 0: */
    if (redeemTokensIn > 0) {
        /*
         * We calculate the exchange rate and the amount of underlying to be redeemed:
         *  redeemTokens = redeemTokensIn
         *  redeemAmount = redeemTokensIn x exchangeRateCurrent
         */
        redeemTokens = redeemTokensIn;
        redeemAmount = mul_ScalarTruncate(exchangeRate, redeemTokensIn);
    } else {
        /*
         * We get the current exchange rate and calculate the amount to be redeemed:
         *  redeemTokens = redeemAmountIn / exchangeRate
         *  redeemAmount = redeemAmountIn
         */
        redeemTokens = div_(redeemAmountIn, exchangeRate);
        redeemAmount = redeemAmountIn;
    }
    require(redeemAmount % expScale == 0, "invalid redeemTokens");

    uint redeemNFTCount = redeemAmount / expScale;

    /* Fail if redeem not allowed */
    uint allowed = comptroller.redeemAllowed(address(this), redeemer, redeemTokens);
    if (allowed != 0) {
        revert RedeemComptrollerRejection(allowed);
    }
```

```
/* Verify market's block number equals current block number */
if (accrualBlockNumber != getBlockNumber()) {
    revert RedeemFreshnessCheck();
}

/* Fail gracefully if protocol has insufficient cash */
if (getNFTsHeld() < redeemNFTCount) {
    revert RedeemTransferOutNotPossible();
}

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

uint redeemInterestAmount = supplyInterestStoredInternal(redeemer);
supplyInterest[redeemer].interestIndex = supplyIndex;

/*
 * We write the previously calculated values into storage.
 *  Note: Avoid token reentrancy attacks by writing reduced supply before external transfer.
 */
totalSupply = totalSupply - redeemTokens;

uint accountTokensNew = accountTokens[redeemer] - redeemTokens;
accountTokens[redeemer] = accountTokensNew;

uint256[] memory nftIds = doNFTTransferOut(redeemer, redeemNFTCount);

if (redeemInterestAmount != 0) {
    supplyInterest[redeemer].interestAccrued = 0;
    interestMarket.collectInterest(redeemer, redeemInterestAmount);
}

/* We emit a Transfer event, and a Redeem event */
emit Transfer(redeemer, address(this), redeemTokens);
emit Redeem(redeemer, redeemNFTCount, redeemTokens, nftIds);

/* We call the defense hook */
comptroller.redeemVerify(address(this), redeemer, redeemAmount, redeemTokens);
```

```
if (accountTokensNew == 0 && borrowBalanceStoredInternal(redeemer) == 0) {
        comptroller.autoExitMarkets(redeemer); // silent failure allowed
    }
}
```

```
function doNFTTransferOut(address to, uint nftCount) virtual internal returns (uint256[] memory nftIds) {
    nftIds = new uint256[](nftCount);
    uint256 nftID;
    uint idx = heldNFTs.length;
    IERC721 underlying_ = IERC721(underlying);
    for(uint i = 0; i < nftCount;) {
        idx--;
        nftID = heldNFTs[idx];
        underlying_.transferFrom(address(this), to, nftID);
        heldNFTs.pop();
        nftIds[i] = nftID;
        unchecked { i++; }
    }
}
```

Commentary from the client:

*“ - By design, all NFTs deposited or withdrawn from the protocol are considered "floor priced". Users should not deposit NFTs they do not want treated as floor priced by the protocol. For a given collection, the NFT you deposit may not be the NFT you eventually withdraw. We document this for users and warn them in the UI.”*

# FNG-3. MISSING MAXIMUM LIMIT ON STALEPRICEDELAY

SEVERITY: Low

PATH: ChainlinkPriceOracle.sol:L150-157

REMEDIATION: add a reasonable maximum limit for the stalePriceDelay

STATUS: fixed

DESCRIPTION:

In the contract **ChainlinkPriceOracle.sol** there is an issue related to the **stalePriceDelay** variable, which is used to determine how long it takes for a Chainlink price feed to be considered stale. Currently, this variable is set to a value of 1 day in comments. However, there is a lack of a maximum limit check when the **setStalePriceDelay** function is called. This means that the admin can set an unbounded value for **stalePriceDelay**, which effectively nullifies the purpose of having a "stale price" check.

```
function setStalePriceDelay(uint _stalePriceDelay) external {
    // Check caller = admin
    if (msg.sender != admin) {
        revert("unauthorized");
    }

    stalePriceDelay = _stalePriceDelay;
}
```

## FNG-19. CUSTOM ERRORS

SEVERITY: Low

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

In various contracts the validation checks are performed using the require function with a reason string.

For example:

```
require(redeemTokensIn == 0 || redeemAmountIn == 0, "one of redeemTokensIn or redeemAmountIn must be zero");
```

We would recommend to replace these with custom errors. This should be done by flipping the check.

For example:

```
require(X == Y, "X is not Y");
```

becomes

```
error XnotY(uint, uint);

if (X != Y)
  revert XnotY(X, Y);
```

The usage of custom errors will save a lot of gas during deployment as well as save on code bytesize of the contract (because strings won't have to be embedded in the code). Furthermore, custom errors are much clearer as they allow for parameter values, making debugging much easier.

## FNG-7. MISSING RECOVER MECHANISMS FOR ETH AND ERC20 TOKENS

SEVERITY: Informational

PATH: CErc20.sol:sweepToken:L134-139

REMEDIATION: recommend adding a mechanism to the CErc20.sol contract to handle the recovery of mistakenly sent ETH, also add a function in the CEther.sol contract that allows for the recovery of accidentally sent ERC20 tokens

STATUS: fixed

DESCRIPTION:

The **sweepToken** function in the **CErc20.sol** contract is designed to recover accidental ERC-20 transfers made to this contract. However, this function does not account for accidentally sending ETH. As a result, if users inadvertently send ETH to this contract, they stand to lose their funds. Similarly, the **CEther.sol** contract lacks a mechanism to reclaim accidentally sent ERC20 tokens, leading to the potential loss of those tokens for users.

```
    /**
     * @notice A public function to sweep accidental ERC-20 transfers to this contract. Tokens are sent to admin
(timelock)
     * @param token The address of the ERC-20 token to sweep
     */
    function sweepToken(EIP20NonStandardInterface token) virtual override external {
        require(msg.sender == admin, "CErc20::sweepToken: only admin can sweep tokens");
        require(address(token) != underlying, "CErc20::sweepToken: can not sweep underlying token");
        uint256 balance = token.balanceOf(address(this));
        token.transfer(admin, balance);
    }
```

# FNG-12. REDUNDANT CODE

SEVERITY: Informational

REMEDIATION: remove all unnecessary or redundant code presented above and in other locations

STATUS: fixed

DESCRIPTION:

There are several contracts with functions and variables that are redundant and would waste deployment gas as they are not used or instantly revert. This also hurts readability.

*CErc721.sol:*

```
function liquidateBorrow(address borrower, uint repayAmount, CTokenInterface cTokenCollateral) override external
returns (uint) {
    revert("unsupported");
}
```

```
function doTransferIn(address from, uint amount) virtual override internal returns (uint) {
    // Read from storage once
    address underlying_ = underlying;
    EIP20NonStandardInterface token = EIP20NonStandardInterface(underlying_);
    uint balanceBefore = EIP20Interface(underlying_).balanceOf(address(this));
    token.transferFrom(from, address(this), amount);

    bool success;
    assembly {
        switch returndatasize()
            case 0 {                    // This is a non-standard ERC-20
                success := not(0)       // set success to true
            }
            case 32 {                   // This is a compliant ERC-20
                returndatacopy(0, 0, 32)
                success := mload(0)     // Set `success = returndata` of override external call
            }
            default {                   // This is an excessively non-compliant ERC-20, revert.
                revert(0, 0)
            }
    }
    require(success, "TOKEN_TRANSFER_IN_FAILED");

    // Calculate the amount that was *actually* transferred
    uint balanceAfter = EIP20Interface(underlying_).balanceOf(address(this));
    return balanceAfter - balanceBefore;   // underflow already checked above, just subtract
}
```

*CErc20.sol:*

```
function liquidateBorrow(address borrower, uint repayAmount, CTokenInterface cTokenCollateral) override external
returns (uint) {
    revert("unsupported");
}
```

*CEther.sol:*

```
function liquidateBorrow(address borrower, uint repayAmount, CTokenInterface cTokenCollateral) override external
returns (uint) {
    revert("unsupported");
}
```

*CToken.sol:*

```
function seize(address liquidator, address borrower, uint seizeTokens) override external /*nonReentrant*/ returns
(uint) {
    revert("unsupported");
}
```

## FNG-13. MISSING EVENT ON CERC721 LIQUIDATION

SEVERITY: Informational

PATH: CErc721.sol:_liquidateBorrow:L427-463

REMEDIATION: we recommend emitting events of crucial functionality and state changes to improve transparency and facilitate protocol integration and off-chain tracking

STATUS: fixed

DESCRIPTION:

The function **_liquidateBorrow** will liquidate an ERC721 loan for a borrower. However, the function does not emit an event upon liquidation, in contrast to liquidations in for example CErc20.

This makes off-chain tracking more difficult and will impair the front-end and user experience.

```
function _liquidateBorrow(address liquidator, address borrower, uint[] memory nftIds) override external nonReentrant
returns (uint) {
    require(msg.sender == address(comptroller), "unauthorized");

    uint assetsExchangeRate = accrueInterest();

    uint repayAmount = nftIds.length * expScale;

    /* Verify market's block number equals current block number */
    if (accrualBlockNumber != getBlockNumber() || assetsExchangeRate == 0) {
        revert LiquidateFreshnessCheck();
    }

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

    /* Fail if repayAmount = 0 */
    if (repayAmount == 0) {
        revert LiquidateCloseAmountIsZero();
    }

    /* Fail if repayBorrow fails */
    uint repayInterest;
    (repayAmount, repayInterest) = repayBorrowFresh(liquidator, borrower, nftIds, 0);

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

    // convert interest value to NFT units
    repayInterest = repayInterest * expScale / assetsExchangeRate;

    // combine in NFT unit terms for seizure calculation
    uint actualRepayAmount = repayAmount + repayInterest;

    return actualRepayAmount;
}
```

## FNG-14. FUNCTION SHOULD BE MARKED EXTERNAL

SEVERITY: Informational

PATH: CErc721.sol:borrowAndInterestBalanceStored:L715-717

REMEDIATION: change it's access modifier to external in favour of code and gas optimisation

STATUS: fixed

DESCRIPTION:

The function **borrowAndInterestBalanceStored** is never called internally, therefore, there is no reason for it to be **public** instead of **external**.

```
function borrowAndInterestBalanceStored(address account) public view returns (uint, uint) {
    return borrowAndInterestBalanceStoredInternal(account);
}
```

# FNG-18. CONDITIONAL EARLY EXIT OPTIMISATION

SEVERITY: Informational

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

There are some conditionals that could be optimised by changing the order of elements by frequency/gas cost to allow for early exits and save gas on each function call.

We have identified the following locations:

1. **CErc721.sol:repayBorrowFresh** on line 381, there is a storage read in the first element and a
2. **CEther.sol:repayBorrowFresh** on line 172;
3. **CToken.sol:repayBorrowFresh** on line 700.

In each of the above mentioned locations, there is a storage read in the first element and a local stack variable in the second element. Since this conditional won't always be triggered, switching the elements would allow for an early exit without a storage read, saving gas on each partial repay.

```
if (accountTokens[borrower] == 0 && accountBorrowsNew == 0) {
    comptroller.autoExitMarkets(borrower); // silent failure allowed
}
```

Change the mentioned code locations to:

```
if (accountBorrowsNew == 0 && accountTokens[borrower] == 0) {
```