Security Audit Report for MoleCity Contracts

Date: Sep 27, 2021

Version: 1.0

Contact: contact@blocksecteam.com

Contents

Report Manifest

Item	Description
Client	MoleCity
Target	MoleCity Contracts

Version History

Version	Date	Description
1.0	Sep 27, 2021	First Release

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high impact security incidents. The team won first place in the 2019 iDash competition (SGX Track). They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The smart contracts that are audited in this report include the following ones.

Contract Name	Github URL
AggregatorV2V3Interface	https://github.com/mole-city/molecity-contract/blob/main/contracts/AggregatorV2V3Interface.sol
BaseJumpRateModelV2	https://github.com/mole-city/molecity-contract/blob/main/contracts/BaseJumpRateModelV2.sol
CarefulMath	https://github.com/mole-city/molecity-contract/blob/main/contracts/CarefulMath.sol
Comptroller	https://github.com/mole-city/molecity-contract/blob/main/contracts/Comptroller.sol
ComptrollerInterface	https://github.com/mole-city/molecity-contract/blob/main/contracts/ComptrollerInterface.sol
ComptrollerStorage	https://github.com/mole-city/molecity-contract/blob/main/contracts/ComptrollerStorage.sol
EIP20Interface	https://github.com/mole-city/molecity-contract/blob/main/contracts/EIP20Interface.sol
EIP20NonStandardInterface	https://github.com/mole-city/molecity-contract/blob/main/contracts/EIP20NonStandardInterface.sol
EIP20Token	https://github.com/mole-city/molecity-contract/blob/main/contracts/EIP20Token.sol
ErrorReporter	https://github.com/mole-city/molecity-contract/blob/main/contracts/ErrorReporter.sol
Exponential	https://github.com/mole-city/molecity-contract/blob/main/contracts/Exponential.sol

Contract Name	Github URL
ExponentialNoError	https://github.com/mole-city/molecity-contract/blob/main/contracts/ExponentialNoError.sol
GovernorAlpha	https://github.com/mole-city/molecity-contract/blob/main/contracts/GovernorAlpha.sol
InterestRateModel	https://github.com/mole-city/molecity-contract/blob/main/contracts/InterestRateModel.sol
JumpRateModel	https://github.com/mole-city/molecity-contract/blob/main/contracts/JumpRateModel.sol
JumpRateModelV2	https://github.com/mole-city/molecity-contract/blob/main/contracts/JumpRateModelV2.sol
LegacyInterestRateModel	https://github.com/mole-city/molecity-contract/blob/main/contracts/LegacyInterestRateModel.sol
LegacyJumpRateModelV2	https://github.com/mole-city/molecity-contract/blob/main/contracts/LegacyJumpRateModelV2.sol
MBep20	https://github.com/mole-city/molecity-contract/blob/main/contracts/MBep20.sol
MBep20Delegate	https://github.com/mole-city/molecity-contract/blob/main/contracts/MBep20Delegate.sol
MBep20Delegator	https://github.com/mole-city/molecity-contract/blob/main/contracts/MBep20Delegator.sol
MBep20Immutable	https://github.com/mole-city/molecity-contract/blob/main/contracts/MBep20Immutable.sol
MBnb	https://github.com/mole-city/molecity-contract/blob/main/contracts/MBnb.sol
MToken	https://github.com/mole-city/molecity-contract/blob/main/contracts/MToken.sol
MTokenInterfaces	https://github.com/mole-city/molecity-contract/blob/main/contracts/MTokenInterfaces.sol
Maximillion	https://github.com/mole-city/molecity-contract/blob/main/contracts/Maximillion.sol

Contract Name	Github URL
Mole	https://github.com/mole-city/molecity-contract/blob/main/contracts/Mole.sol
MoleCityOracle	https://github.com/mole-city/molecity-contract/blob/main/contracts/MoleCityOracle.sol
MoleDaoVoteRelay	https://github.com/mole-city/molecity-contract/blob/main/contracts/MoleDaoVoteRelay.sol
MoleLock	https://github.com/mole-city/molecity-contract/blob/main/contracts/MoleLock.sol
MoleThrower	https://github.com/mole-city/molecity-contract/blob/main/contracts/MoleThrower.sol
PriceOracle	https://github.com/mole-city/molecity-contract/blob/main/contracts/PriceOracle.sol
SafeMath	https://github.com/mole-city/molecity-contract/blob/main/contracts/SafeMath.sol
SimplePriceOracle	https://github.com/mole-city/molecity-contract/blob/main/contracts/SimplePriceOracle.sol
Unitroller	https://github.com/mole-city/molecity-contract/blob/main/contracts/Unitroller.sol
WhitePaperInterestRateModel	https://github.com/mole-city/molecity-contract/blob/main/contracts/WhitePaperInterestRateModel.sol

The commit SHA value of the initial reviewed files is:
59e9c281b2d931f57242535116edfeb6ea464310
The commit hash after checking in the fixes reported by this audit is:
6d66b0ef7f412532ee6ea090065176b50dfcecfb

1.2 Disclaimer

This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, this report does not constitute personal investment advice or a personal recommendation.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data Flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Access control

- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [1] and Common Weakness Enumeration [2]. Accordingly, the severity measured in this report are classified into four categories: **High**, **Medium**, **Low** and **Undetermined**.

[1] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[2] https://cwe.mitre.org/

Chapter 2 Findings

In total, we have identified **6 potential issues** and 2 additional recommendations, as follows:

- High Risk: 2
- Medium Risk: 2
- Low Risk: 2

ID	Severity	Description	Category
1	Medium	Semantic inconsistency of the `claimMole` function	Software Security
2	High	The function `emergencyWithdraw` has a risk of re-entrancy attack	Software Security
3	Low	The function `getBalance` is called but has no implementation	Software Security
4	Medium	The Mole distribution mechanism has a potential risk of being attacked by flashloan	DeFi Security
5	High	Potential re-entrancy from tokens with callback mechanism	Defi Security
6	Low	The function `getVotes` has a risk to be manipulated by flashloan	Defi Security
7	-	Single price oracle	Recommendation
8	-	The design of governance is not decentralized	Recommendation

The details are provided in the following sections.

2.1 Software Security

2.1.1 Semantic inconsistency of the `claimMole` function

Status Confirmed and fixed.

Description

There are three public functions: `claimMole` in line 1242, 1252, and 1265 of the contract `Comptroller`. However, only the first one (line 1243) invokes the internal function `updateContributorRewards`, which causes the semantic inconsistency.

```
1238   /**
1239    * @notice Claim all the mole accrued by holder in all markets
1240    * @param holder The address to claim MOLE for
1241    */
1242  function claimMole(address holder) public {
1243      updateContributorRewards(holder);
1244      return claimMole(holder, allMarkets);
1245  }
1246
1247  /**
1248   * @notice Claim all the mole accrued by holder in the specified markets
```

```
1249     * @param holder The address to claim MOLE for
1250     * @param mTokens The list of markets to claim MOLE in
1251     */
1252    function claimMole(address holder, MToken[] memory mTokens) public {
1253        address[] memory holders = new address[](1);
1254        holders[0] = holder;
1255        claimMole(holders, mTokens, true, true);
1256    }
1257
1258    /**
1259     * @notice Claim all mole accrued by the holders
1260     * @param holders The addresses to claim MOLE for
1261     * @param mTokens The list of markets to claim MOLE in
1262     * @param borrowers Whether or not to claim MOLE earned by borrowing
1263     * @param suppliers Whether or not to claim MOLE earned by supplying
1264     */
1265    function claimMole(address[] memory holders, MToken[] memory mTokens, bool borrowers, bool
            suppliers) public {
1266        for (uint i = 0; i < mTokens.length; i++) {
1267            MToken mToken = mTokens[i];
1268            require(markets[address(mToken)].isListed, "market must be listed");
1269            if (borrowers == true) {
1270                Exp memory borrowIndex = Exp({mantissa: mToken.borrowIndex()});
1271                updateMoleBorrowIndex(address(mToken), borrowIndex);
1272                for (uint j = 0; j < holders.length; j++) {
1273                    distributeBorrowerMole(address(mToken), holders[j], borrowIndex, true);
1274                }
1275            }
1276            if (suppliers == true) {
1277                updateMoleSupplyIndex(address(mToken));
1278                for (uint j = 0; j < holders.length; j++) {
1279                    distributeSupplierMole(address(mToken), holders[j], true);
1280                }
1281            }
1282        }
1283    }
```

Impact Users invoke different implementations of the function `claimMole` may cause different contracts' states.

Suggestion Removing the function `updateContributorRewards` invocation in line 1243 or adding the function `updateContributorRewards` invocation in line 1253 and line 1266.

2.1.2 The function `emergencyWithdraw` has a risk of re-entrancy attacks

Status Confirmed and fixed.

Description

There exists a classical re-entrancy problem in function `emergencyWithdraw`. Once the `lptoken` has the callback mechanism (such as ERC777 token), attackers can steal all `lptokens` in the contract `MoleThrower` by re-entering this function `emergencyWithdraw` before or after transferring tokens.

```
749     // Withdraw without caring about rewards. EMERGENCY ONLY.
```

```
function emergencyWithdraw(uint256 _pid) public {

    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];

    uint256 amount = user.amount;

    // transfer LP tokens to user
    pool.lpToken.safeTransfer(address(msg.sender), amount);

    pool.totalDeposit = pool.totalDeposit.sub(user.amount);
    // update user info
    user.amount = 0;
    user.rewardDebt = 0;

    emit EmergencyWithdraw(msg.sender, _pid, amount);
}
```

Impact `Lptokens` with callback mechanism can be stolen by re-entrancy attacks.

Suggestion Re-ordering the codes by adhering to the check-effects-interactions pattern.

2.1.3 The function `getBalance` is called but has no implementation

Status Confirmed and fixed.

Description

In the function `getVotes`, an external call `moleThrower.getBlance` is made. However, there is no related implementation in the `MoleThrower` contract.

```
/**
* @notice query the user's pledge balance in several pools, and return the sum of each pledge
    balance multiplied by weight
* @param _user The address of the user
* @return Total number of votes
*/
function getVotes(address _user) external view returns (uint256) {
  uint totalVote = 0;
  for (uint256 i = 0; i < moleLocks.length; i++) {
    address lockAddress = moleLocks[i];
    uint period = voteWeights[lockAddress].period;
    uint weight = voteWeights[lockAddress].weight;
    uint balance = 0;
    if (period == 0) { //0 deposit and 0 fetch
      MoleThrowerInterface moleThrower = MoleThrowerInterface(lockAddress);
      balance = moleThrower.getBalance(0, _user);
    } else {
      MoleLockInterface moleLock = MoleLockInterface(lockAddress);
      uint256 releaseTime = moleLock.releaseTime();
      //Expired molelock do not count
      if (block.timestamp < releaseTime) {
        balance = moleLock.balanceOf(_user);
      }
    }
```

```
    totalVote = add(totalVote, mul(balance, weight));
  }
  return totalVote;
}
```

Impact Since the function `getBalance` can not be found, the variable `balance` always be zero. That causes the error calculation of vote power.

Suggestion Add an correct implementation of `getBalance` in the contract `MoleThrower`.

2.2 DeFi Security

2.2.1 The Mole distribution mechanism has a potential risk of being attacked by flashloan

Status Confirmed and fixed.

Description

MoleCity has a mechanism to distribute Moles to users at a certain speed. The contract `Comptroller` updates proportions of Moles distributed in each market after each borrowing and repaying, because it determines the proportion according to the states of each market. For example, the market with the highest utilization can be allocated to the most Moles. Meanwhile, MoleCity distributes Moles to borrowers according to their debts. Since flashloan can temporarily affect the markets' states of MoleCity, there is a potential risk of being attacked when the price of Mole is high enough.

Particularly, an attacker borrows huge amounts of assets through flashloan and then deposits them into MoleCity as collateral. Due to the huge amount of collateralized assets, the attacker can borrow out a huge amount of assets from MoleCity, which results in 1)*first* a large percentage of Moles being allocated to the involved market; 2) and *then* a large percentage of Moles being allocated to the attacker. After that, the attacker repays the loans, redeems the collateralized assets, and repays the flashloan. If the value of allocated Mole is higher than the repayment interests, the attacker can profit and almost monopolize Moles distributed by MoleCity at each block in this way.

```
function refreshMoleSpeedsInternal() internal {
  MToken[] memory allMarkets_ = allMarkets;

  for (uint i = 0; i < allMarkets_.length; i++) {
    MToken mToken = allMarkets_[i];
    Exp memory borrowIndex = Exp({mantissa: mToken.borrowIndex()});
    updateMoleSupplyIndex(address(mToken));
    updateMoleBorrowIndex(address(mToken), borrowIndex);
  }

  Exp memory totalUtility = Exp({mantissa: 0});
  Exp[] memory utilities = new Exp[](allMarkets_.length);
  for (uint i = 0; i < allMarkets_.length; i++) {
    MToken mToken = allMarkets_[i];
    if (markets[address(mToken)].isMoleed) {
      Exp memory assetPrice = Exp({mantissa: oracle.getUnderlyingPrice(mToken)});
      Exp memory utility = mul_(assetPrice, mToken.totalBorrows());
      utilities[i] = utility;
```

```
      totalUtility = add_(totalUtility, utility);
    }
  }

  for (uint i = 0; i < allMarkets_.length; i++) {
    MToken mToken = allMarkets[i];
    uint newSpeed = totalUtility.mantissa > 0 ? mul_(moleRate, div_(utilities[i], totalUtility))
         : 0;
    moleSpeeds[address(mToken)] = newSpeed;
    emit MoleSpeedUpdated(mToken, newSpeed);
  }
}
```

```
function borrowVerify(address mToken, address borrower, uint borrowAmount) external {
  // Shh - currently unused
  mToken;
  borrower;
  borrowAmount;

  // Shh - we don't ever want this hook to be marked pure
  if (false) {
    maxAssets = maxAssets;
  }

  refreshMoleSpeedsInternal();

}
```

```
function repayBorrowVerify(
address mToken,
address payer,
address borrower,
uint actualRepayAmount,
uint borrowerIndex) external {
  // Shh - currently unused
  mToken;
  payer;
  borrower;
  actualRepayAmount;
  borrowerIndex;

  // Shh - we don't ever want this hook to be marked pure
  if (false) {
    maxAssets = maxAssets;
  }

  refreshMoleSpeedsInternal();

}
```

Impact There is a potential risk to interfering with the Mole distribution mechanism.

Suggestion Manually updating the proportions of Moles distributed in each market, which can avoid the impact of flashloan.

2.2.2 Potential reentrancy risks from tokens with callback mechanism

Status Confirmed. **The project owner ensures that assets with callback mechanism (like ERC777) will not be supported, which avoids the issues.**

Description

In MoleCity, every user has a bill that records his/her liquidity and shortfall. Once the liquidity is bigger than zero, the user can borrow underlying assets. This bill can be calculated in the function `getHypotheticalAccountLiquidityInternal`. And shown in the following codes, the function `getHypotheticalAccountLiquidityInternal` traverses all pools in MoleCity and then calculates a sum of liquidity for a user.

```
/**
 * @notice Determine what the account liquidity would be if the given amounts were redeemed/borrowed
 * @param mTokenModify The market to hypothetically redeem/borrow in
 * @param account The account to determine liquidity for
 * @param redeemTokens The number of tokens to hypothetically redeem
 * @param borrowAmount The amount of underlying to hypothetically borrow
 * @dev Note that we calculate the exchangeRateStored for each collateral mToken using stored data,
 *  without calculating accumulated interest.
 * @return (possible error code,
            hypothetical account liquidity in excess of collateral requirements,
 *          hypothetical account shortfall below collateral requirements)
 */
function getHypotheticalAccountLiquidityInternal(
    address account,
    MToken mTokenModify,
    uint redeemTokens,
    uint borrowAmount) internal view returns (Error, uint, uint) {

    AccountLiquidityLocalVars memory vars; // Holds all our calculation results
    uint oErr;

    // For each asset the account is in
    MToken[] memory assets = accountAssets[account];
    for (uint i = 0; i < assets.length; i++) {
        MToken asset = assets[i];

        // Read the balances and exchange rate from the mToken
        (oErr, vars.mTokenBalance, vars.borrowBalance, vars.exchangeRateMantissa) = asset.getAccountSnapshot(account);
        if (oErr != 0) { // semi-opaque error code, we assume NO_ERROR == 0 is invariant between upgrades
            return (Error.SNAPSHOT_ERROR, 0, 0);
        }
        vars.collateralFactor = Exp({mantissa: markets[address(asset)].collateralFactorMantissa});
        vars.exchangeRate = Exp({mantissa: vars.exchangeRateMantissa});

        // Get the normalized price of the asset
        vars.oraclePriceMantissa = oracle.getUnderlyingPrice(asset);
```

```
        if (vars.oraclePriceMantissa == 0) {
            return (Error.PRICE_ERROR, 0, 0);
        }
        vars.oraclePrice = Exp({mantissa: vars.oraclePriceMantissa});

        // Pre-compute a conversion factor from tokens -> ether (normalized price value)
        vars.tokensToDenom = mul_(mul_(vars.collateralFactor, vars.exchangeRate), vars.oraclePrice);

        // sumCollateral += tokensToDenom * mTokenBalance
        vars.sumCollateral = mul_ScalarTruncateAddUInt(vars.tokensToDenom, vars.mTokenBalance, vars.sumCollateral);

        // sumBorrowPlusEffects += oraclePrice * borrowBalance
        vars.sumBorrowPlusEffects = mul_ScalarTruncateAddUInt(vars.oraclePrice, vars.borrowBalance, vars.sumBorrowPlusEffects);

        // Calculate effects of interacting with mTokenModify
        if (asset == mTokenModify) {
            // redeem effect
            // sumBorrowPlusEffects += tokensToDenom * redeemTokens
            vars.sumBorrowPlusEffects = mul_ScalarTruncateAddUInt(vars.tokensToDenom, redeemTokens, vars.sumBorrowPlusEffects);

            // borrow effect
            // sumBorrowPlusEffects += oraclePrice * borrowAmount
            vars.sumBorrowPlusEffects = mul_ScalarTruncateAddUInt(vars.oraclePrice, borrowAmount, vars.sumBorrowPlusEffects);
        }
    }

    // These are safe, as the underflow condition is checked first
    if (vars.sumCollateral > vars.sumBorrowPlusEffects) {
        return (Error.NO_ERROR, vars.sumCollateral - vars.sumBorrowPlusEffects, 0);
    } else {
        return (Error.NO_ERROR, 0, vars.sumBorrowPlusEffects - vars.sumCollateral);
    }
}
```

The function `borrow` invokes the function `borrowInternal` in the contract `MToken.sol` and eventually calls an internal function `borrowFresh`. The codes (line 786 - 792) in the function `borrowFresh` fails to use the "Check-Effects-Interactions" pattern. Even if there is a modifier `nonReentrant` in function `borrowInternal`, attackers can still re-enter other `mToken` contracts, once the underlying token has callback mechanism. Since the market states are updated (line 789 - 791) after the `doTransferOut`, `Comptroller` will get a wrong result. By explioting this, attackers can get additional assets from other pools.

```
710  /**
711    * @notice Sender borrows assets from the protocol to their own address
712    * @param borrowAmount The amount of the underlying asset to borrow
713    * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
714    */
715  function borrowInternal(uint borrowAmount) internal nonReentrant returns (uint) {
```

```
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but we still want to log the fact that an
                attempted borrow failed
            return fail(Error(error), FailureInfo.BORROW_ACCRUE_INTEREST_FAILED);
        }
        // borrowFresh emits borrow-specific logs on errors, so we don't need to
        return borrowFresh(msg.sender, borrowAmount);
    }
```

```
    /**
    * @notice Users borrow assets from the protocol to their own address
    * @param borrowAmount The amount of the underlying asset to borrow
    * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
    */
  function borrowFresh(address payable borrower, uint borrowAmount) internal returns (uint) {
      /* Fail if borrow not allowed */
      uint allowed = comptroller.borrowAllowed(address(this), borrower, borrowAmount);
      if (allowed != 0) {
          return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.BORROW_COMPTROLLER_REJECTION,
              allowed);
      }

      /* Verify market's block number equals current block number */
      if (accrualBlockNumber != getBlockNumber()) {
          return fail(Error.MARKET_NOT_FRESH, FailureInfo.BORROW_FRESHNESS_CHECK);
      }

      /* Fail gracefully if protocol has insufficient underlying cash */
      if (getCashPrior() < borrowAmount) {
          return fail(Error.TOKEN_INSUFFICIENT_CASH, FailureInfo.BORROW_CASH_NOT_AVAILABLE);
      }

      BorrowLocalVars memory vars;

      /*
       * We calculate the new borrower and total borrow balances, failing on overflow:
       *  accountBorrowsNew = accountBorrows + borrowAmount
       *  totalBorrowsNew = totalBorrows + borrowAmount
       */
      (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
      if (vars.mathErr != MathError.NO_ERROR) {
          return failOpaque(Error.MATH_ERROR, FailureInfo.
              BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
      }

      (vars.mathErr, vars.accountBorrowsNew) = addUInt(vars.accountBorrows, borrowAmount);
      if (vars.mathErr != MathError.NO_ERROR) {
          return failOpaque(Error.MATH_ERROR, FailureInfo.
              BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
      }

      (vars.mathErr, vars.totalBorrowsNew) = addUInt(totalBorrows, borrowAmount);
```

```
    if (vars.mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR, FailureInfo.
            BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
    }

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

    /*
     * We invoke doTransferOut for the borrower and the borrowAmount.
     *  Note: The mToken must handle variations between ERC-20 and ETH underlying.
     *  On success, the mToken borrowAmount less of cash.
     *  doTransferOut reverts if anything goes wrong, since we can't be sure if side effects
        occurred.
     */
    doTransferOut(borrower, borrowAmount);

    /* We write the previously calculated values into storage */
    accountBorrows[borrower].principal = vars.accountBorrowsNew;
    accountBorrows[borrower].interestIndex = borrowIndex;
    totalBorrows = vars.totalBorrowsNew;

    /* We emit a Borrow event */
    emit Borrow(borrower, borrowAmount, vars.accountBorrowsNew, vars.totalBorrowsNew);

    /* We call the defense hook */
    // unused function
    comptroller.borrowVerify(address(this), borrower, borrowAmount);

    return uint(Error.NO_ERROR);
}
```

Impact Attackers can borrow more valuable assets from MoleCity than collateralized assets.

Suggestion Re-ordering the codes to follow the pattern "Check-Effects-Interactions", like moving `doTransferOut` after updating market states.

2.2.3 The function `getVotes` has a risk to be manipulated by flashloan

Status Confirmed and fixed.

Description

The function `getVotes` relies on the staking balance of user (line 71). Attacker can stake huge amount lptoken by flashloan to obtain voting power in one transaction.

```
57  /**
58   * @notice query the user's pledge balance in several pools, and return the sum of each pledge
         balance multiplied by weight
59   * @param _user The address of the user
60   * @return Total number of votes
61   */
62  function getVotes(address _user) external view returns (uint256) {
63      uint totalVote = 0;
```

```
        for (uint256 i = 0; i < moleLocks.length; i++) {
            address lockAddress = moleLocks[i];
            uint period = voteWeights[lockAddress].period;
            uint weight = voteWeights[lockAddress].weight;
            uint balance = 0;
            if (period == 0) { //0 deposit and 0 fetch
                MoleThrowerInterface moleThrower = MoleThrowerInterface(lockAddress);
                balance = moleThrower.getBalance(0, _user);
            } else {
                MoleLockInterface moleLock = MoleLockInterface(lockAddress);
                uint256 releaseTime = moleLock.releaseTime();
                //Expired molelock do not count
                if (block.timestamp < releaseTime) {
                    balance = moleLock.balanceOf(_user);
                }
            }
            totalVote = add(totalVote, mul(balance, weight));
        }
        return totalVote;
    }
```

Impact Someone can obtain additional voting power by flashloan to manipulate voting result.

Suggestion Adding logics to calculate voting power by both staking amount and staking time.

2.3 Additional Recommendations

2.3.1 Single price oracle problem

Status Acknowledged

Description

The price oracle of `MoleCity` is `ChainLink` now. It must assume the price from `ChainLink` is solid and trustable.

Impact If the price oracle of `ChainLink` has been compromised, the assets in MoleCity are at risk of being stolen.

Suggestion It will be better to use multiple oracles and do cross-validation.

2.3.2 The design of governance is not decentralized

Status Acknowledged

Description

The contract `GovernorAlpha` only records user's proposals. The execution of these proposals are not automatic and decentralized, which means executing these proposals requires the project party send transactions manually.

Impact The design of governance is not decentralized.

Suggestion Using decentralized design of governance.

Chapter 3 Conclusion

In this audit, we have analyzed the business logic, the design, and the implementation of the MoleCity Contracts. Overall, the current code base is well-structured and implemented. Most of the issues that we pointed out have been fixed, and the project party ensures that assets with callback mechanism (like ERC777) will not be supported. Meanwhile, as previously disclaimed, this report does not give any warranties on discovering all security issues of the smart contracts. We appreciate any constructive feedback or suggestions.