Competitive Security Assessment

Tonka_Finance_Lending

Feb 2nd, 2024

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

- Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
- Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
- Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
- Verify the code base is compliant with the most up-to-date industry standards and security best practices.
- Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

Overview

Project Detail

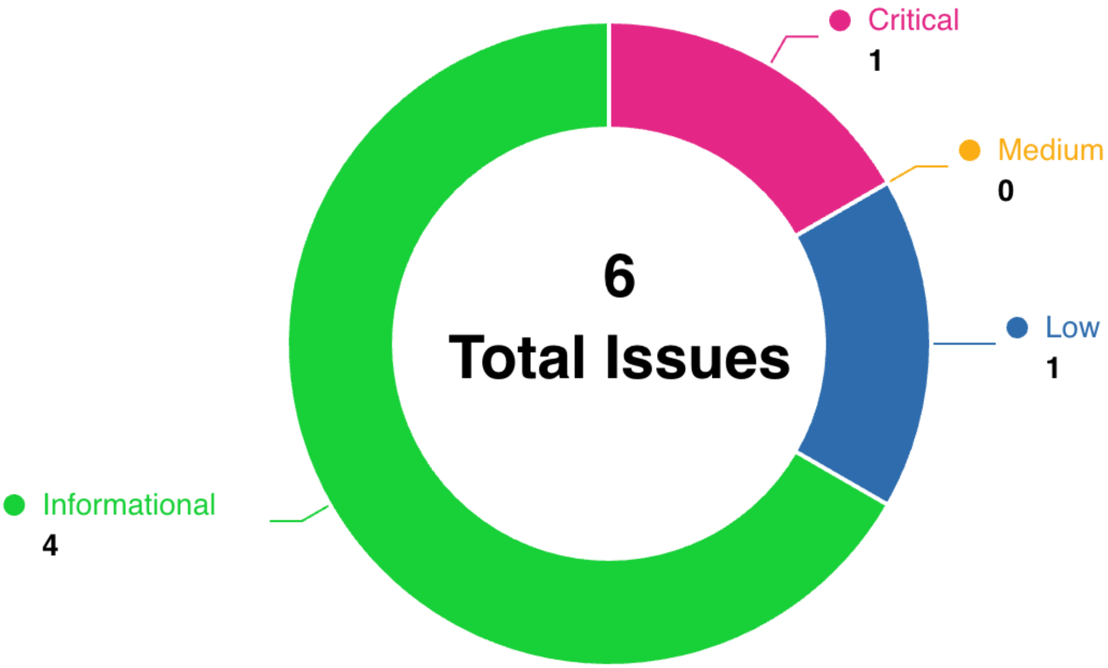
Project Name	Tonka_Finance_Lending
Platform & Language	Solidity
Codebase	• https://github.com/Tonka-Finance/Tonka-Contracts • audit commit - 005e1857d7d0b08d0e70ca4b1314878e2375512e • final commit - 9a0c2907def1e15a92e94953f8f9294d5656b300
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

Audit Scope

File	SHA256 Hash
contracts/Money-Market/TToken.sol	a5bf8b8ad2c9cd7ac598bb123e6f4b62c313c4b3426b7ae5e8713a698cd8d8f6
contracts/Money-Market/Comptroller.sol	788393e9e03f11107cd403d6af6839f3ab2df305257f74a375e9af872c75d807
contracts/Money-Market/Lens/TonkaLens.sol	899344ab9e9ac9e132cd0bb4c87ecf1fc47c859e68ec7107c97bd5151769f02e
contracts/Money-Market/TErc20Delegator.sol	09c2ef6d442a5a1c9cb76f6f77b1d9c1b2415932378b82cd850fc225ac656813
contracts/Money-Market/ErrorReporter.sol	dee6d886c725ebfa0463ef8733d9bf6a9c8cf1476673dd4a078329f38d44d67a
contracts/Money-Market/Exponential.sol	77e1821fcacde2356892aedc303fe540307b945fbf80cef3dbc37da31be5874a
contracts/Money-Market/TTokenInterfaces.sol	8a3de767b4096a5bd9af2a147e1f00f63d5cb23d954636e69a1f1349f576048a
contracts/Money-Market/TErc20.sol	3b92803b46e79eb2c0bf1aaa35507545755a9f9b97243a40dc9641c95db17b47
contracts/Money-Market/Unitroller.sol	0fff715ba967b38cae659d497264e92e6b1b37b0fad3e155d8667c38264251de
contracts/Money-Market/ComptrollerInterface.sol	34f920320cd7ef9cf758883d009491d9941eadea4696647d5bce60f5170c76c5
contracts/Money-Market/SimpleInterestRateModel.sol	40485cbda42610343a7df077fdbf6d5a3d0e29f933e5b2c45e6634989250436c
contracts/Money-Market/ComptrollerStorage.sol	219d86d62cd381082c32cfa381305e17ab4d3ab6fe3a9497ca2d7b52a989d390
contracts/Money-Market/SimplePriceOracle.sol	20d520a4883763d8214272a857376533ac76c79751f7d5dc8f471f189d1483e9
contracts/Money-Market/TErc20Delegate.sol	5c90aec036d1679f5fa7df8bcea94e9c1e83f2c359d7c45dbed6cdd13613fe32
contracts/Money-Market/EIP20Interface.sol	f9f680a37b68b5f85b0462943397f54c7eb04cdd0239e4129e545866c1a1aae5

contracts/Money-Market/EIP20NonStandardInterface.sol	**f4e8e637b3c1e8eb8f685877ed545d15e77d5a8053d49ab57c53668ed3f1180b**
contracts/Money-Market/InterestRateModel.sol	**be5837928ba0bd049de30542f02c567c821a64af848275e064fc02c170f29eff**
contracts/Money-Market/PriceOracle.sol	**b4556f67ce3b890685be645149df049a3167635524b45caa9883ad19f169bd19**

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
TFL-1	Reentrancy Risk	Logical	Critical	Fixed	Yaodao, danielt
TFL-2	Logical issue of function `exchangeRateStoredInternal()`	Logical	Critical	Declined	Yaodao
TFL-3	Lack of check the maxAssets	Logical	Low	Fixed	danielt
TFL-4	ComptrollerInterface::function parameters refers address of tokens as cToken instead of tToken	Code Style	Informational	Fixed	ravikiran_web3, n16h7m4r3

TFL-5	Return value not checked	Code Style	Informational	Fixed	Yaodao
TFL-6	`TonkaLens::tTokenMetadata()` - incorrect comparison can result in returning incorrect underlyingAssetAddress	Code Style	Informational	Fixed	ravikiran_web3, n16h7m4r3
TFL-7	Duplicate return statements in `SimpleInterestRateModel.utilizationRate()` function	Logical	Informational	Fixed	Yaodao, 0xac, n16h7m4r3

TFL-1:Reentrancy Risk

Category	Severity	Client Response	Contributor
Logical	**Critical**	**Fixed**	**Yaodao, danielt**

Code Reference

- code/contracts/Money-Market/TToken.sol#L625-L633
- code/contracts/Money-Market/TToken.sol#L667-L721
- code/contracts/Money-Market/TToken.sol#L707-L715

```
625:doTransferOut(redeemer, vars.redeemAmount);
626:
627:        /* We write previously calculated values into storage */
628:        totalSupply = vars.totalSupplyNew;
629:        accountTokens[redeemer] = vars.accountTokensNew;
630:
631:        /* We emit a Transfer event, and a Redeem event */
632:        emit Transfer(redeemer, address(this), vars.redeemTokens);
633:        emit Redeem(redeemer, vars.redeemAmount, vars.redeemTokens);

667:function borrowFresh(address payable borrower, uint borrowAmount) internal returns (uint) {
668:        /* Fail if borrow not allowed */
669:        uint allowed = comptroller.borrowAllowed(address(this), borrower, borrowAmount);
670:        if (allowed != 0) {
671:            return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.BORROW_COMPTROLLER_REJECTION, allowed);
672:        }
673:
674:        /* Verify market's block number equals current block number */
675:        if (accrualBlockNumber != getBlockNumber()) {
676:            return fail(Error.MARKET_NOT_FRESH, FailureInfo.BORROW_FRESHNESS_CHECK);
677:        }
678:
679:        /* Fail gracefully if protocol has insufficient underlying cash */
680:        if (getCashPrior() < borrowAmount) {
681:            return fail(Error.TOKEN_INSUFFICIENT_CASH, FailureInfo.BORROW_CASH_NOT_AVAILABLE);
682:        }
683:
684:        BorrowLocalVars memory vars;
685:
686:        /*
687:         * We calculate the new borrower and total borrow balances, failing on overflow:
688:         *  accountBorrowsNew = accountBorrows + borrowAmount
689:         *  totalBorrowsNew = totalBorrows + borrowAmount
690:         */
691:        vars.accountBorrows = borrowBalanceStoredInternal(borrower);
692:
693:        vars.accountBorrowsNew = vars.accountBorrows + borrowAmount;
```

```

        vars.totalBorrowsNew = totalBorrows + borrowAmount;

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

        /*
         * We invoke doTransferOut for the borrower and the borrowAmount.
         *  Note: The tToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the tToken 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 */
        comptroller.borrowVerify(address(this), borrower, borrowAmount);

        return uint(Error.NO_ERROR);
    }

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);
```

Description

Yaodao : The following codes in the functions redeemFresh() and borrowFresh() do not meet the Checks-Effects-Interactions pattern.

```
        doTransferOut(redeemer, vars.redeemAmount);

        /* We write previously calculated values into storage */
        totalSupply = vars.totalSupplyNew;
        accountTokens[redeemer] = vars.accountTokensNew;

        /* We emit a Transfer event, and a Redeem event */
        emit Transfer(redeemer, address(this), vars.redeemTokens);
        emit Redeem(redeemer, vars.redeemAmount, vars.redeemTokens);

        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);
```

It only has a reentrancy lock as there is no lock at the controller level, only the TToken level.

If the `tToken` is an ERC777 protocol, the reentrancy can happen at function levels of an ERC777-based contract, i.e., multiple function calls that are triggered by the hook mechanism of ERC777.

This issue is possible to happen with all compound forks, but Compound is not affected as they do not list tokens with callback functionality.

danielt : The `borrowFresh` function allows users to borrow assets from the protocol to their own address. However, the `borrowFresh` function firstly transfers tokens to the borrower, then updates `accountBorrows`, which incurs risks for ERC777-type assets. If the borrowed asset is an ERC777-type asset, the transfer of the ERC777 token will call `_callTokensReceived` so that the attacker can call `borrowFresh()` again in reentrancy. Though there is a `nonReentrant` modifier on the `borrowInternal` function, it can only protect function-level re-entrancy attacks but can not prevent the cross-contract reentrancy attack.

The vulnerability is fixed in the commit 1da68862... by the compound protocol.

One of the attacks on it happened before:

C.R.E.A.M. Finance Post Mortem: AMP Exploit: *The AMP token contract implements ERC777, which has the _callPostTransferHooks hook that triggers tokensReceived() function that was implemented by the recipient. The reentrancy opportunity related to ERC-777-style transfer hooks allowed the exploiter to nest a second borrow() function inside the token transfer() before the initial borrow() was updated.*

Recommendation

Yaodao : Recommend using the Checks-Effects-Interactions pattern and understanding the security limitations of the forking compound.
danielt : Recommend updating the `accountBorrows` first, later than transferring the asset to the borrower, to match the CEI pattern, to prevent the cross-contract reentrancy issue, like what did in the commit 1da68862...

Client Response

Fixed: use CEI pattern commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/c57034cb33f4fc2a0c81ec0e23b02eb47b7d4b1c

TFL-2:Logical issue of function `exchangeRateStoredInternal()`

Category	Severity	Client Response	Contributor
Logical	Critical	Declined	Yaodao

Code Reference

- code/contracts/Money-Market/TToken.sol#L313-L336

```
313:function exchangeRateStoredInternal() internal view returns (uint) {
314:        uint _totalSupply = totalSupply;
315:        if (_totalSupply == 0) {
316:            /*
317:             * If there are no tokens minted:
318:             *  exchangeRate = initialExchangeRate
319:             */
320:            return initialExchangeRateMantissa;
321:        } else {
322:            /*
323:             * Otherwise:
324:             *  exchangeRate = (totalCash + totalBorrows - totalReserves) / totalSupply
325:             */
326:            uint totalCash = getCashPrior();
327:            uint cashPlusBorrowsMinusReserves;
328:            Exp memory exchangeRate;
329:
330:            cashPlusBorrowsMinusReserves = totalCash + totalBorrows - totalReserves;
331:
332:            exchangeRate = getExp(cashPlusBorrowsMinusReserves, _totalSupply);
333:
334:            return exchangeRate.mantissa;
335:        }
336:    }
```

Description

Yaodao : In the aforementioned line, the formula for the calculation of `exchangeRate` is as follows after `tToken` is minted:

exchangeRate = (totalCash + totalBorrows - totalReserves)/ totalSupply

```
    function exchangeRateStoredInternal() internal view returns (uint) {
        uint _totalSupply = totalSupply;
        if (_totalSupply == 0) {
            /*
             * If there are no tokens minted:
             *  exchangeRate = initialExchangeRate
             */
            return initialExchangeRateMantissa;
        } else {
            /*
             * Otherwise:
             *  exchangeRate = (totalCash + totalBorrows - totalReserves) / totalSupply
             */
            uint totalCash = getCashPrior();
            uint cashPlusBorrowsMinusReserves;
            Exp memory exchangeRate;

            cashPlusBorrowsMinusReserves = totalCash + totalBorrows - totalReserves;

            exchangeRate = getExp(cashPlusBorrowsMinusReserves, _totalSupply);

            return exchangeRate.mantissa;
        }
    }
```

In solidity, division calculations have truncation problems. The totalSupply will be `1` and `exchangeRate` will be much large than initialExchangeRate in case the last user redeems `(accountTokens[redeemer] - 1)` tToken.

As a result, the `exchangeRate` would be extremely large.

When the value of `exchangeRate` is much larger than `initialExchangeRate`, the user can mint `tTokens` well above normal values, and then the value of `exchangeRate` will be normal with the interest generating. In other words, the users can use this arbitrage to take away the underlying tokens in this pool.

For example, the user can mint the amount of 1e8 `tToken` with one underlying token in case `exchangeRate` = 1/1e8.

1. For the 1st tx: transfer 8,097.251216520237142235 underlying tokens and mint 404,862.56082601 tTokens(exchangeRate is normal)
2. For the 2nd tx: redeem 1000 underlying tokens and burn 49,999.99999999 tTokens(exchangeRate is normal)
3. For the 3rd tx: redeem 7,097.251216520237142234 underlying tokens and burn 354,862.56082601 tTokens

For now:

total underlying tokens transfer: 8,097.251216520237142235

total underlying tokens redeem: 1000 + 7,097.251216520237142234 = 8,097.251216520237142234

total underlying tokens still in contract: 0.000000000000000001

total tTokens mint: 404,862.56082601

total tTokens burn: 49,999.99999999 + 354,862.56082601 = 404862.56082600

total tTokens remain: 0.00000001

Thus, the exchangeRate is abnormal.

4. For the 4th tx: As the exchangeRate is abnormal, transfer 8,097.251216520237142234 underlying tokens will mint 80,972,512,165,202.37142234 tTokens(1e10 of the underlying tokens, which is determined by the decimals)

Recommendation

Yaodao : Recommend using the following solutions to help mitigate this issue:

1. Adding reasonable upper and lower boundaries to replace the return value when the `exchangeRate` is un-reasonable big or small,
2. Adding a new contract that can only call `mint()` but can't call `redeem()` to supply reasonable amounts of the underlying token to the pool.

Client Response

Declined,This situation is highly unlikely to occur in real-world scenarios and is limited to the last user redeeming tTokens. Furthermore, the time it takes for the exchangeRate to return to normal would be significantly long. To further mitigate the risk, we will deposit additional underlying assets to prevent the risk.

TFL-3:Lack of check the maxAssets

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	danielt

Code Reference

- code/contracts/Money-Market/Comptroller.sol#L952-L963

```
function _setMaxAssets(uint newMaxAssets) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_MAX_ASSETS_OWNER_CHECK);
        }

        uint oldMaxAssets = maxAssets;
        maxAssets = newMaxAssets;
        emit NewMaxAssets(oldMaxAssets, newMaxAssets);

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

Description

danielt : The external function `_setMaxAssets` allows the admin to update the `maxAssets`, which restricts the number of the market a borrower can be used as "asset in" for liquidation calculation and borrowing. If it is zero, it will result in the borrower being unable to borrow. So, it is recommended to add a non-zero check on the `maxAssets`.

Recommendation

danielt : Recommend adding a non-zero check on the `maxAssets`.

Client Response

Fixed: add zero check commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/11620051115e51f6b16c2c97372eee2a7b60de19

TFL-4:ComptrollerInterface::function parameters refers address of tokens as cToken instead of tToken

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	ravikiran_web3, n16h7m4r3

Code Reference

- code/contracts/Money-Market/ComptrollerInterface.sol#L4-L92

```
abstract contract ComptrollerInterface {
    /// @notice Indicator that this is a Comptroller contract (for inspection)
    bool public constant isComptroller = true;

    /*** Assets You Are In ***/

    function enterMarkets(address[] calldata cTokens) external virtual returns (uint[] memory);

    function exitMarket(address cToken) external virtual returns (uint);

    /*** Policy Hooks ***/

    function mintAllowed(address cToken, address minter, uint mintAmount) external virtual returns (uint);

    function mintVerify(address cToken, address minter, uint mintAmount, uint mintTokens) external virtual;

    function redeemAllowed(address cToken, address redeemer, uint redeemTokens) external virtual returns (uint);

    function redeemVerify(address cToken, address redeemer, uint redeemAmount, uint redeemTokens) external virtual;

    function borrowAllowed(address cToken, address borrower, uint borrowAmount) external virtual returns (uint);

    function borrowVerify(address cToken, address borrower, uint borrowAmount) external virtual;

    function repayBorrowAllowed(
        address cToken,
        address payer,
        address borrower,
        uint repayAmount
    ) external virtual returns (uint);

    function repayBorrowVerify(
        address cToken,
        address payer,
        address borrower,
        uint repayAmount,
        uint borrowerIndex
    ) external virtual;

    function liquidateBorrowAllowed(
        address cTokenBorrowed,
        address cTokenCollateral,
        address liquidator,
        address borrower,
        uint repayAmount
    ) external virtual returns (uint);

    function liquidateBorrowVerify(
        address cTokenBorrowed,
        address cTokenCollateral,
        address liquidator,
        address borrower,
        uint repayAmount,
        uint seizeTokens
    ) external virtual;

    function seizeAllowed(
        address cTokenCollateral,
        address cTokenBorrowed,
        address liquidator,
        address borrower,
        uint seizeTokens
    ) external virtual returns (uint);

    function seizeVerify(
        address cTokenCollateral,
        address cTokenBorrowed,
        address liquidator,
        address borrower,
        uint seizeTokens
    ) external virtual;

    function transferAllowed(
        address cToken,
        address src,
        address dst,
        uint transferTokens
    ) external virtual returns (uint);

    function transferVerify(address cToken, address src, address dst, uint transferTokens) external virtual;

    /*** Liquidity/Liquidation Calculations ***/

    function liquidateCalculateSeizeTokens(
        address cTokenBorrowed,
        address cTokenCollateral,
        uint repayAmount
    ) external view virtual returns (uint, uint);
}
```

Description

ravikiran_web3 : Incorrect naming convention for parameters in ComptrollerInterface.
n16h7m4r3 : In the interface contract `ComptrollerInterface` the `cToken` is referred instead of `tToken`.

Recommendation

ravikiran_web3 : Update the names to standard convention across the project.
n16h7m4r3 : Consider updating the variable names in the contract.

Client Response

Fixed: c to t commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/e8bf4ef78cd7bdd11cd47ec91ecd0623139c4e82

TFL-5:Return value not checked

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	Yaodao

Code Reference

- code/contracts/Money-Market/Comptroller.sol#L1016

```
1016:tToken.isTToken(); // Sanity check to make sure its really a TToken
```

Description

Yaodao : The return value of an external call is not checked.

```
    function _supportMarket(TToken tToken) external returns (uint) {
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SUPPORT_MARKET_OWNER_CHECK);
        }

        if (markets[address(tToken)].isListed) {
            return fail(Error.MARKET_ALREADY_LISTED, FailureInfo.SUPPORT_MARKET_EXISTS);
        }

        tToken.isTToken(); // Sanity check to make sure its really a TToken

        Market storage newMarket = markets[address(tToken)];
        newMarket.isListed = true;
        newMarket.collateralFactorMantissa = 0;

        emit MarketListed(tToken);

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

Recommendation

Yaodao : Recommend adding “require” statement for isTToken:

```
require(tToken.isTToken();,”This is not a TToken contract!”);
```

Client Response

Fixed: add require for tToken.isTToken() commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/89623211f93cfa07abd7c2ac01cb63425cbe7964

TFL-6: `TonkaLens::tTokenMetadata()` - incorrect comparison can result in returning incorrect underlyingAssetAddress

Category	Severity	Client Response	Contributor
Code Style	**Informational**	**Fixed**	**ravikiran_web3, n16h7m4r3**

Code Reference

- code/contracts/Money-Market/Lens/TonkaLens.sol#L28-L42
- code/contracts/Money-Market/Lens/TonkaLens.sol#L35
- code/contracts/Money-Market/Lens/TonkaLens.sol#L88

```
28:function tTokenMetadata(TToken tToken) public returns (TTokenMetadata memory) {
29:        uint exchangeRateCurrent = tToken.exchangeRateCurrent();
30:        Comptroller comptroller = Comptroller(address(tToken.comptroller()));
31:        (bool isListed, uint collateralFactorMantissa) = comptroller.markets(address(tToken));
32:        address underlyingAssetAddress;
33:        uint underlyingDecimals;
34:
35:        if (compareStrings(tToken.symbol(), "cETH")) {
36:            underlyingAssetAddress = address(0);
37:            underlyingDecimals = 18;
38:        } else {
39:            TErc20 tErc20 = TErc20(address(tToken));
40:            underlyingAssetAddress = tErc20.underlying();
41:            underlyingDecimals = EIP20Interface(tErc20.underlying()).decimals();
42:        }

35:if (compareStrings(tToken.symbol(), "cETH")) {

88:if (compareStrings(tToken.symbol(), "cETH")) {
```

Description

ravikiran_web3 : The tokenMetaData compares the symbol of token with cETH for native token, while it should be **tETH.** This incorrect comparison will result in returning wrong underlying asset address.

```
    function tTokenMetadata(TToken tToken) public returns (TTokenMetadata memory) {
          uint exchangeRateCurrent = tToken.exchangeRateCurrent();
          Comptroller comptroller = Comptroller(address(tToken.comptroller()));
          (bool isListed, uint collateralFactorMantissa) = comptroller.markets(address(tToken));
          address underlyingAssetAddress;
          uint underlyingDecimals;

          if (compareStrings(tToken.symbol(), "cETH")) {
              underlyingAssetAddress = address(0);
              underlyingDecimals = 18;
          } else {
```

n16h7m4r3 : In the functions `tTokenMetadata()` and `tTokenBalances()` the tToken symbol is compared with `cETH`, and would return the structure `TTokenMetadata` and `TTokenBalances` with incorrect values.

Recommendation

ravikiran_web3 : Update the comparing string to **tETH**
n16h7m4r3 : The symbol for `tToken` is `tETH`, consider updating the values in the contract.

Client Response

Fixed: modify cETH to tETH commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/d9b1761d71773e30560e44df59e4cb9a07bffd80 and https://github.com/Tonka-Finance/Tonka-Contracts/commit/9a0c2907def1e15a92e94953f8f9294d5656b300

TFL-7:Duplicate return statements in `SimpleInterestRateModel.utilizationRate()` function

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	Yaodao, 0xac, n16h7m4r3

Code Reference

- code/contracts/Money-Market/SimpleInterestRateModel.sol#L53-L54
- code/contracts/Money-Market/SimpleInterestRateModel.sol#L53
- code/contracts/Money-Market/SimpleInterestRateModel.sol#L54

```
53:return (borrows * 1e18) / (cash + borrows - reserves);
54:        return (borrows * 1e18) / (cash + borrows - reserves);

53:return (borrows * 1e18) / (cash + borrows - reserves);

54:return (borrows * 1e18) / (cash + borrows - reserves);
```

Description

Yaodao : The following codes are duplicated, and we should remove the duplicated codes.

```
    function utilizationRate(uint cash, uint borrows, uint reserves) public pure returns (uint) {
        // Utilization rate is 0 when there are no borrows
        if (borrows == 0) {
            return 0;
        }

        return (borrows * 1e18) / (cash + borrows - reserves);
        return (borrows * 1e18) / (cash + borrows - reserves);
    }
```

0xac : There are two identical return statements in `SimpleInterestRateModel.utilizationRate()` function, and the latter will not be executed, so this is a redundant code.

n16h7m4r3 : In the provided Solidity code snippet, there is a redundant and unreachable return statement after the first one. In Solidity, as in many programming languages, once a return statement is executed, the control flow exits the function, and any subsequent code is not executed.

Recommendation

Yaodao : Recommend removing the duplicated codes.
0xac : It is recommended to delete one `return` statement and keep one.
n16h7m4r3 : Consider removing the return statement.

Client Response

Fixed: remove the duplicated codes commit: https://github.com/Tonka-Finance/Tonka-Contracts/commit/e7fb43ebe439cd6d85b240a9a4700ddd7ca7da94

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3’s prior written consent in each instance.

This report is not an “endorsement” or “disapproval” of any particular project or team. This report is not an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3’s position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.