Security Audit Report for Torches Smart Contracts

Date: September 29, 2022

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.09.30 12:00:17 +08'00'

Contents

Report Manifest

Item	Description
Client	Torches Finance
Target	Torches Smart Contracts

Version History

Version	Date	Description
1.0	September 29, 2022	First Release

Chapter 1 Introduction

1.1 About Target Contracts

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

The target of this audit includes two repositories of the Torches project, i.e., `torches-protocol` and `torches-dao`. The former is the main lending protocol implementation forked from the Compound Protocol, while the latter is the governance and reward component for the Torches project. Note that two contracts (i.e., `TorchesAssetOracle.sol` and `TorchesPriceOracle.sol`) in the `torches-protocol` repository are out of the audit scope.

Repo Name	Github URL
torches-protocol	https://github.com/TorchesFinance/torches-protocol
torches-dao	https://github.com/TorchesFinance/torches-dao

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
torches-protocol	Version 1a	0bee06cf08e407b84813d7bbee3d25e1496164aa
	Version 2a	c1c4b074817ed44d2683843e957c00a45fbe0398
torches-dao	Version 1b	e785cb8208ae2a8a34c7799a0b5cb2668e0ded68
	Version 2b	e810174a34616c27c4daa239e6f9e6f2d8790964

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. 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.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

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
- Permission management
- 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]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

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

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

Chapter 2 Findings

In total, we find **eight** potential issues. Besides, we also have **three** notes.

- High Risk: 1
- Medium Risk: 6
- Low Risk: 1
- Note: 3

ID	Severity	Description	Category	Status
1	Medium	Potential access control problem in the `VotingEscrow` contract	Software Security	Fixed
2	Medium	Potential price manipulation on the reward allocation speed	DeFi Security	Fixed
3	Medium	Delayed updates of voting powers in the `LiquidityGaugeV3` contract	DeFi Security	Undetermined
4	Medium	Insufficient check of prices	DeFi Security	Acknowledged
5	Medium	Insufficient checks in the `TMLPDelegate` contract	DeFi Security	Fixed
6	Medium	Accouting update problem in the `LiquidityGaugeV3` contract	DeFi Security	Fixed
7	Low	Potential inconsistent epoch length in the `LiquidityGaugeV3` contract	DeFi Security	Acknowledged
8	High	Token balance manipulation in the `CToken` contract	DeFi Security	Fixed
9	-	Modification on the market model	Note	
10	-	Price oracle assumptions	Note	
11	-	About the absurdly high borrow rates	Note	

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential access control problem in the `VotingEscrow` contract

Severity Medium

Status Fixed in `Version 2b`

Introduced by `Version 1b`

Description

In the `torches-dao` project, there is a potential access control bug in the `deposit_for` function of the `VotingEscrow` contract. As a premise, the `VotingEscrow` contract is designed for users locking their TOR tokens (i.e., the reward and governance token of the Torches Protocol) in exchange for the voting power. However, anyone is able to lock the TOR tokens of other users (i.e., the victims in the following context) through the `deposit_for` function, if the following requirements are met:

- The victims have TOR tokens.
- The victims have approved enough allowance of the TOR tokens to the `VotingEscrow` contract.

- The victims have created locks in the `VotingEscrow` contract, and the locks haven't expired.

Under such circumstance, anyone could invoke the `deposit_for` function to lock any outstanding TOR token holdings of the victims.

```
@external
@nonreentrant('lock')
def deposit_for(_addr: address, _value: uint256):
    """
    @notice Deposit `_value` tokens for `_addr` and add to the lock
    @dev Anyone (even a smart contract) can deposit for someone else, but
         cannot extend their locktime and deposit for a brand new user
    @param _addr User's wallet address
    @param _value Amount to add to user's lock
    """
    _locked: LockedBalance = self.locked[_addr]

    assert _value > 0 # dev: need non-zero value
    assert _locked.amount > 0, "No existing lock found"
    assert _locked.end > block.timestamp, "Cannot add to expired lock. Withdraw"

    self._deposit_for(_addr, _value, 0, self.locked[_addr], DEPOSIT_FOR_TYPE)
```

Listing 2.1: VotingEscrow.vy

Note that the above problem may become more serious due to the reward-claiming mechanism of this project. Specifically, in the `claim_rewards_for` function of the `RewardHelper` contract, anyone could arbitrarily claim rewards for other users. Though the rewards are always claimed into the correct accounts, the insufficient access control may cause a problem of calculating the reward amounts.

```
@external
def claim_rewards_for(_addr: address, _gauges: address[10]):
    """
    @notice Claim available reward tokens for `_addr`
    @param _addr Address to claim for
    @param _gauges Gauge addresses to claim rewards
    """
    assert _addr != ZERO_ADDRESS, "invalid parameter"

    controller: address = self.controller
    minter: address = self.minter
    for gauge in _gauges:
        # check gauge is added
        if gauge != ZERO_ADDRESS and Controller(controller).gauge_types(gauge) >= 0:
            Minter(minter).mint_for(gauge, _addr)
            LiquidityGauge(gauge).claim_rewards(_addr)
```

Listing 2.2: RewardHelper.vy

Hence the `claim_rewards_for` function would affect the checkpoints recorded in the `LiquidityGaugeV3` contract. If the reward-claiming procedure can be controlled by others, the amount of the rewards may be different and may cause losses to the users. As a result, anyone can lock not only the TOR tokens held by someone but also the rewards can be claimed.

Impact Arbitrarily claiming rewards for any users may affect the reward calculation and cause potential losses to the users.

Impact TOR tokens can be locked in the `VotingEscrow` contract by any other users.

Suggestion Allow users to set a whitelist for delegating token locking.

2.2 DeFi Security

2.2.1 Potential price manipulation on the reward allocation speed

Severity Medium

Status Fixed in `Version 2b`

Introduced by `Version 1b`

Description In the `torches-dao` project, there exists a price manipulation problem in the `_checkpoint` function of the `LiquidityGaugeV3` contract. The project provides incentives to the holders of `tToken`s by adopting a mechanism similar to the Curve DAO project. Every time the function is invoked, a checkpoint is recorded for allocating the rewards.

Specifically, every balance change of `tTokens` would trigger a hook calling to the corresponding `LiquidityGaugeV3` contract and accumulate the reward in TOR tokens. This hook updates the reward allocation points (points for short) based on the following two-step calculation:

1. The reward allocation speed for a certain time t is calculated as $r(t) = p * s(t)/WEEK$, where p is a system parameter, $s(t)$ is the total supply of `tToken` **recorded in the first checkpoint of a week**, $WEEK$ is the total seconds in a week (604800).
2. The points of a certain user is calculated as $I_u = \int \frac{r(t)b_u(t)}{s(t)}$, where $b_u(t)$ is the balance of the user at time t, and $s(t)$ is the current total supply of `tToken`.

In the actual implementation, the reward allocation speed $r(t)$ is only updated once per week. If there is no checkpoint in a whole week, then the $r(t)$ of the previous period is used. Therefore, there exists a path that malicious actors can use the flashloan to manipulate the reward allocation rates:

1. At the beginning of a week, borrowing the flashloan, and depositing underlying tokens to mint the corresponding `tToken`. This process would increase the `totalSupply` of `tToken`.
2. Since the `new_rate` calculation happens after updating the `totalSupply` of `tToken`, creating a checkpoint by calling functions like `user_checkpoint`. The creation of the checkpoint would **increase the recorded reward rate** thus further biases the reward allocation speed.
3. Redeeming `tToken` deposited to repay the flashloan.

The above steps may lead to biased reward allocation speed which would increase the amount of rewards for the current epoch.

```
@internal
def _checkpoint(addr: address):
    """
    @notice Checkpoint for a user
    @param addr User address
    """
    _point_period: int128 = self.point_period
    _point_period_timestamp: uint256 = self.point_period_timestamp[_point_period]
    _point_integrate_inv_supply: uint256 = self.point_integrate_inv_supply[_point_period]

    rate: uint256 = self.point_rate
    prev_epoch: uint256 = self.point_current_epoch_time
    new_rate: uint256 = rate
    next_epoch: uint256 = prev_epoch + WEEK

    if block.timestamp > next_epoch:
        new_totalSupply: uint256 = ERC20(self.lp_token).totalSupply()
        if new_totalSupply > 0:
            new_rate = self.point_proportion * new_totalSupply / WEEK
        self.point_current_epoch_time = next_epoch
        self.point_rate = new_rate

    # Update integral of 1/supply
    if block.timestamp > _point_period_timestamp and not self.is_killed:
        prev_week_time: uint256 = _point_period_timestamp
        week_time: uint256 = min((_point_period_timestamp + WEEK) / WEEK * WEEK, block.timestamp)
        _totalSupply: uint256 = self.lpTotalSupply

        for i in range(500):
            dt: uint256 = week_time - prev_week_time
            if _totalSupply > 0:
                if next_epoch >= prev_week_time and next_epoch < week_time:
                    # If we went across epoch, apply the rate
                    # of the first epoch until it ends, and then the rate of
                    # the last epoch.
                    _point_integrate_inv_supply += rate * (next_epoch - prev_week_time) /
                        _totalSupply
                    rate = new_rate
                    _point_integrate_inv_supply += rate * (week_time - next_epoch) / _totalSupply
```

Listing 2.3: LiquidityGaugeV3.vy

```
@external
def user_checkpoint(addr: address) -> bool:
    """
    @notice Record a checkpoint for ‘addr‘
    @param addr User address
    @return bool success
    """
    assert (msg.sender == addr) or (msg.sender == self.minter) # dev: unauthorized
    self._checkpoint(addr)
    self._checkpoint_dao(addr)
    self._checkpoint_rewards(addr, False, ZERO_ADDRESS)
    self._update_liquidity_limit(addr, self._balance_of(addr), self.totalSupply)
    return True
```

Listing 2.4: LiquidityGaugeV3.vy

Impact The calculation of the reward allocation speed can be manipulated by using the flashloan.

Suggestion Check the rate updating mechanism.

2.2.2 Delayed updates of voting powers in the `LiquidityGaugeV3` contract

Severity Medium

Status Undetermined

Introduced by `Version 1b`

Description In the `LiquidityGaugeV3` contract of the `torches-dao` project, the reward distribution calculation involves the `working_balance` variable of users. A user's `working_balance` is calculated as follows:

$$B = \min(I, 0.4I + 0.6S\frac{w}{W})$$

Here I is the current reward points of the user, S is the total reward points recorded in a variable named `working_supply`, while w and W represent the voting power the user holds and the total voting power recorded in the `VotingEscrow` contract, respectively.

If users do not update their voting power in the `VotingEscrow` contract, the corresponding w and W will decrease at the same speed, so the ratio of the user's voting power to the total voting power (i.e., w/W) remains the same. If there are other users locking TOR tokens in the `VotingEscrow` contract, the total voting power would increase, which means the ratio a certain user would decrease. However, if the `_update_liquidity_limit` function is not invoked, the ratio will not be modified.

In summary, if users lock TOR tokens in the `VotingEscrow` contract, it would reduce the voting power ratio of all other users, but this decrease is not updated into the `LiquidityGaugeV3` contract in time. It may lead to incorrect calculation of the reward amounts.

```
@internal
def _update_liquidity_limit(addr: address, l: uint256, L: uint256):
    """
    @notice Calculate limits which depend on the amount of CRV token per-user.
            Effectively it calculates working balances to apply amplification
            of CRV production by CRV
    @param addr User address
    @param l User's amount of liquidity (LP tokens)
    @param L Total amount of liquidity (LP tokens)
    """
    # To be called after totalSupply is updated
    _voting_escrow: address = self.voting_escrow
    voting_balance: uint256 = ERC20(_voting_escrow).balanceOf(addr)
    voting_total: uint256 = ERC20(_voting_escrow).totalSupply()

    lim: uint256 = l * TOKENLESS_PRODUCTION / 100
    if voting_total > 0:
        lim += L * voting_balance / voting_total * (100 - TOKENLESS_PRODUCTION) / 100

    lim = min(l, lim)
    old_bal: uint256 = self.working_balances[addr]
    self.working_balances[addr] = lim
    _working_supply: uint256 = self.working_supply + lim - old_bal
    self.working_supply = _working_supply

    log UpdateLiquidityLimit(addr, l, L, lim, _working_supply)
```

Listing 2.5: LiquidityGaugeV3.vy

Impact Delayed updates of the voting power ratio of the users may cause incorrect reward distribution.

Suggestion N/A

2.2.3 Insufficient check of prices

Severity Medium

Status Acknowledged

Introduced by `Version 1a`

Description In the `ChainlinkAdaptor` contract of the `torches-protocol` project, the `getPrice` function does not check the delay of the retrieved price. In this function, the `latestRoundData` function (of the underlying Chainlink-compliant price oracle contract) is invoked. Though the `latestRoundData` function returns the timestamp of the price update, this timestamp is ignored in the `getPrice` function, thus it is possible that expired prices are used.

Besides, the original Chainlink implementation would provide a valid range `(minAnswer, maxAnswer)` for each price feed. However, the `ChainlinkAdaptor` does not check the validity of the price accordingly.

```
function getPrice(ChainlinkAggregatorV3Interface priceSource) public view returns (uint256) {
  (,int256 answer,,,) = priceSource.latestRoundData();
  if (answer > 0) {
      return uint256(answer);
  } else {
      return 0;
  }
}
```

Listing 2.6: ChainlinkAdaptor.sol

Impact The contract may get expired or invalid prices.

Suggestion Check the delay and the valid range for each retrieved price.

Feedback from the Developers All price oracles used in the `torches-protocol` is in a separate project named `torches-oracle` (which is out of the audit scope). The price oracles in this project take the proper price range into consideration and implement related checks on the prices. The price oracles receives prices from off-chain agents and validate the prices with MojitoSwap TWAP prices and WitNet prices on the KCC chain (which is referred to as the “anchor price”). The updated price would be checked in a 5% range with anchor prices. There are also off-chain monitoring service to monitor prices and update timestamps.

2.2.4 Insufficient checks in the `TMLPDelegate` contract

Severity Medium

Status Fixed in `Version 2a`

Introduced by `Version 1a`

Description In the `torches-protocol` project, there is a special type of `tToken` implemented in the `TMLPDelegate` contract. This type of `tToken` is used to provide a wrapped token for the LP tokens from the MojitoSwap project. When depositing into this `tToken`, the provided underlying LP tokens are deposited into the Farming module of MojitoSwap for accumulating rewards. However, there are some assumptions that are not checked in the `TMLPDelegate` contract:

1. The `TMLPDelegate` contract would retrieve and record the amount of the reward tokens from MojitoSwap and distribute them to all the users. So the `rewardsTokens` list should not contain the underlying LP token, otherwise, the rewards accounting in the contract would be incorrect.
2. The `MasterChefV2` contract of MojitoSwap assumes that the Pool 0 that accepts the deposits and distributes the rewards both in the `MJT` token. Therefore, the `pid` parameter cannot be zero in the `TMLPDelegate` contract.
3. The `TMLPDelegate` contract inherits the `CErc20Delegate` contract and thus provides the `sweepToken` function. The original `sweepToken` function only forbids claiming the underlying token, however, in the context of the `TMLPDelegate` contract, the claiming of the reward tokens should also be forbidden to avoid accounting errors.

```
function _becomeImplementation(bytes memory data) public {
    super._becomeImplementation(data);

    (address poolAddress_, uint pid_) = abi.decode(data, (address, uint));
    mojitoPool = MasterChefV2(poolAddress_);
    MasterChefV2.PoolInfo memory poolInfo = mojitoPool.poolInfo(pid_);
    require(underlying == address(poolInfo.lpToken), "mismatch underlying");

    pid = pid_;

    if (rewardsTokens.length == 0) {
        rewardsTokens.push(address(mojitoPool.mojito()));
    }
    if (address(poolInfo.rewarder) != address(0)) {
        address rewardToken = address(IRewarder(poolInfo.rewarder).rewardToken());
        bool exist = false;
        for (uint8 i = 0; i < rewardsTokens.length; i++) {
            if (rewardsTokens[i] == rewardToken) {
                exist = true;
                break;
            }
        }
        if (!exist) rewardsTokens.push(rewardToken);
    }

    // Approve moving our LP into the pool contract.
    EIP20Interface(underlying).approve(poolAddress_, uint(-1));
    EIP20Interface(rewardsTokens[0]).approve(poolAddress_, uint(-1));
}
```

Listing 2.7: TMLPDelegate.sol

Impact Incorrect assumption may bring unexpected behaviors.

Suggestion Implement correct checks in the `TMLPDelegate` contract.

2.2.5 Accouting update problem in the `LiquidityGaugeV3` contract

Severity Medium

Status Fixed in Version 2b

Introduced by Version 1b

Description The LiquidityGaugeV3 contract in the torches-dao project is implemented as a fork from the Curve Protocol with some modification. However, there exist some interdependencies of the variables that could cause accounting update problem due to the modification. Specifically, there are two functions, _checkpoint_dao and _checkpoint_rewards, that calculate rewards based on a variable named working_supply. If working_supply is updated without calling the corresponding checkpoint function, it would cause accounting update problem in the LiquidityGaugeV3 contract, as follows:

1. In the claim_rewards function, the _update_liquidity_limit function is called without calling the _checkpoint_dao function. Because the former function updates working_supply, it would cause accounting problems.

```
@external
@nonreentrant('lock')
def claim_rewards(_addr: address = msg.sender, _receiver: address = ZERO_ADDRESS):
    """
    @notice Claim available reward tokens for `_addr`
    @param _addr Address to claim for
    @param _receiver Address to transfer rewards to - if set to
                     ZERO_ADDRESS, uses the default reward receiver
                     for the caller
    """
    if _receiver != ZERO_ADDRESS:
        assert _addr == msg.sender # dev: cannot redirect when claiming for another user
    self._checkpoint_rewards(_addr, True, _receiver)

    # update user's point amount and working balance
    self._checkpoint(_addr)
    self._update_liquidity_limit(_addr, self._balance_of(_addr), self.totalSupply)
```

Listing 2.8: LiquidityGaugeV3.vy

2. Similarly, the user_checkpoint function calls the _update_liquidity_limit function, but does not call the _checkpoint_rewards function. Hence it has the similar accounting problem.

```
@external
def user_checkpoint(addr: address) -> bool:
    """
    @notice Record a checkpoint for `addr`
    @param addr User address
    @return bool success
    """
    assert (msg.sender == addr) or (msg.sender == self.minter) # dev: unauthorized
    self._checkpoint(addr)
    self._checkpoint_dao(addr)
    self._update_liquidity_limit(addr, self._balance_of(addr), self.totalSupply)
    return True
```

Listing 2.9: LiquidityGaugeV3.vy

Impact Incorrect handling of the interdependencies of state variables may cause accounting problems.

Suggestion Revise the code.

2.2.6 Potential inconsistent epoch length in the `LiquidityGaugeV3` contract

Severity Low

Status Acknowledged

Introduced by `Version 1b`

Description In the `torches-dao` project, there exists a potential inconsistency in the `_checkpoint_dao` function of the `LiquidityGaugeV3` contract. Specifically, the epoch retrieved from `RewardPolicyMaker` (at line 302) is assumed to be the length of `WEEK` (which is the total seconds in a week, i.e., 604,800 seconds). However, in the `RewardPolicyMaker` contract, the epoch length may not be fixed. This inconsistency may bring unexpected results.

```
280 @internal
281 def _checkpoint_dao(addr: address):
282     """
283     @notice Checkpoint interest for a user
284     @param addr User address
285     """
286     _period: int128 = self.period
287     _period_time: uint256 = self.period_timestamp[_period]
288     _integrate_inv_supply: uint256 = self.integrate_inv_supply[_period]
289 
290     _epoch: uint256 = RewardPolicyMaker(self.reward_policy_maker).epoch_at(block.timestamp)
291     if _period_time == 0:
292         _period_time = RewardPolicyMaker(self.reward_policy_maker).epoch_start_time(_epoch)
293 
294     # Update integral of 1/supply
295     if block.timestamp > _period_time and not self.is_killed:
296         _working_supply: uint256 = self.working_supply
297         _controller: address = self.controller
298         Controller(_controller).checkpoint_gauge(self)
299         prev_week_time: uint256 = _period_time
300 
301         for i in range(500):
302             _epoch = RewardPolicyMaker(self.reward_policy_maker).epoch_at(prev_week_time)
303             week_time: uint256 = RewardPolicyMaker(self.reward_policy_maker).epoch_start_time(
                    _epoch + 1)
304             week_time = min(week_time, block.timestamp)
305 
306             dt: uint256 = week_time - prev_week_time
307             w: uint256 = Controller(_controller).gauge_relative_weight(self, prev_week_time / WEEK
                    * WEEK)
```

Listing 2.10: LiquidityGaugeV3.vy

Impact Inconsistent settings of the parameters may bring unexpected results.

Suggestion Make the configuration and the calculation consistent.

Feedback from the Developers We would make sure that the `epoch_length` parameter in the `RewardPolicyMaker` contract is properly set to one week in seconds.

2.2.7 Token balance manipulation in the CToken contract

Severity High

Status Fixed in Version 2a

Introduced by Version 1a

Description The torches-protocol project is a fork from the Compound Protocol. It uses the CToken and Comptroller contracts as its base system, and modifies the market model.

As reported by the OpenZeppelin for the original Compound Protocol audit [1], there is a potential vulnerability that a malicious operator can manipulate the underlying balance for the CToken contract. Unfortunately, the torches-protocol project still suffers from this problem.

As shown in the following code segment, the underlying token balance manipulation affects the calculation of the exchange rate. The calculation formula for the exchange rate is as follows:

$$exchangeRate = \frac{totalCash + totalBorrows - totalReserves}{totalSupply}$$

A potential attack scenario happens when totalSupply of CToken is very low. In this scenario, a malicious actor can transfer a large amount of the underlying tokens to the contract. Specifically, totalCash is increased, but the remain factors of the formula is unchanged. Therefore, the exchangeRate is miscalculated and can result in wrong number of tokens minted for a illiquid or new market.

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

            (mathErr, cashPlusBorrowsMinusReserves) = addThenSubUInt(totalCash, totalBorrows,
                totalReserves);
            if (mathErr != MathError.NO_ERROR) {
                return (mathErr, 0);
            }

            (mathErr, exchangeRate) = getExp(cashPlusBorrowsMinusReserves, _totalSupply);
            if (mathErr != MathError.NO_ERROR) {
                return (mathErr, 0);
            }
```

[1] Reference: https://blog.openzeppelin.com/compound-comprehensive-protocol-audit/

```
        return (MathError.NO_ERROR, exchangeRate.mantissa);
    }
}
```

Listing 2.11: CToken.sol

Impact The underlying token balance manipulation can affect the calculation of the exchange rate and may cause unexpected behaviors.

Suggestion Revise the code accordingly.

2.3 Note

2.3.1 Modification on the market model

Description The `torches-protocol` project is a lending protocol forked from the Compound protocol with some modification on its economic model. Torches added a system parameter named **credit limit** for different user addresses which could affect the behavior of the lending protocol:

1. If the credit limit for an account is 2^{256}, then the hypothetical liquidity is considered $2^{256}-1$, regardless the current collateral and borrow state of the account.
2. If the credit limit for an account is larger than zero, then it is not allowed to liquidate this account, and the collateral amount for this account is set to this credit limit, which would affect the calculation of the hypothetical liquidity.

In brief, improper settings of the credit limit parameters would cause centralization risk and affect the correct functionality of the lending protocol.

2.3.2 Price oracle assumptions

Description According to the feedback of the developers, the `torches-protocol` project has special assumptions on the price oracle, as follows:

1. All the tokens used by this project have a consistent token decimal, i.e., 18.
2. The price sources of the `ChainlinkAdaptor` contracts are in a separate project maintained by the Torches team, and use exactly the same data format and precision.

This audit strictly follows these assumptions. Note that, it might cause severe price oracle problems if the project maintainers/operators break these assumptions.

2.3.3 About the absurdly high borrow rates

Description The original implementation of the `CToken` contract has a path that would cause the entire contract locked down if the borrow rate is absurdly high in the `accrueInterest` function.

```
function accrueInterest() public returns (uint) {
    /* Remember the initial block number */
    uint currentBlockNumber = getBlockNumber();
    uint accrualBlockNumberPrior = accrualBlockNumber;

    /* Short-circuit accumulating 0 interest */
```

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

        /* Read the previous values out of storage */
        uint cashPrior = getCashPrior();
        uint borrowsPrior = totalBorrows;
        uint reservesPrior = totalReserves;
        uint borrowIndexPrior = borrowIndex;

        /* Calculate the current borrow interest rate */
        uint borrowRateMantissa = interestRateModel.getBorrowRate(cashPrior, borrowsPrior,
            reservesPrior);
        require(borrowRateMantissa <= borrowRateMaxMantissa, "borrow rate is absurdly high");
```

Listing 2.12: CToken.sol

To address this problem, in the modified version of this contract in the `torches-protocol` project (i.e., the `tToken` contract), there is a parameter called `borrowCap` limiting the total amount of `tToken` borrowed. If this parameter is properly set, this check would pass and the `tToken` contract would function correctly. Otherwise, the check would fail and the `tToken` contract would still be locked down.