# SMART CONTRACT AUDIT REPORT

for

# Topshelf Protocol

Prepared By: Patrick Liu

PeckShield

March 3, 2022

## Document Properties

| | |
|---|---|
| Client | Topshelf Protocol |
| Title | Smart Contract Audit Report |
| Target | Topshelf |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Stephen Bie, Patrick Liu, Xuxian Jiang |
| Reviewed by | Patrick Liu |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | March 3, 2022 | Xuxian Jiang | Final Release |
| 1.0-rc1 | January 15, 2021 | Xuxian Jiang | Release Candidate #1 |

## Contact

For more information about this document and its contents, please contact PeckShield Inc.



| | |
|---|---|
| Name | Patrick Liu |
| Phone | +86 183 5897 7782 |
| Email | contact@peckshield.com |

# Contents

# 1 Introduction

Given the opportunity to review the **Topshelf** design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

## 1.1 About Topshelf

`Topshelf` is a decentralized borrowing protocol that allows you to draw low-interest loans against a variety of tokens used as collateral. Loans are paid out in an synthetic USD pegged stablecoin and need to maintain a minimum (configurable) collateral ratio. In addition to the collateral, the loans are secured by a `stability pool` containing the synthetic stablecoin and by fellow borrowers collectively acting as guarantors of last resort. Initially forked from `Liquity`, `Topshelf` makes a number of extensions by supporting an `ERC20` as collateral, streaming protocol token emissions, as well as improved capital efficiency with flashloans.

The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Topshelf

| Item | Description |
|---|---|
| Name | Topshelf Protocol |
| Type | Ethereum Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | March 3, 2022 |

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/bartend3r/topshelf-contracts.git (9e9cbd2)

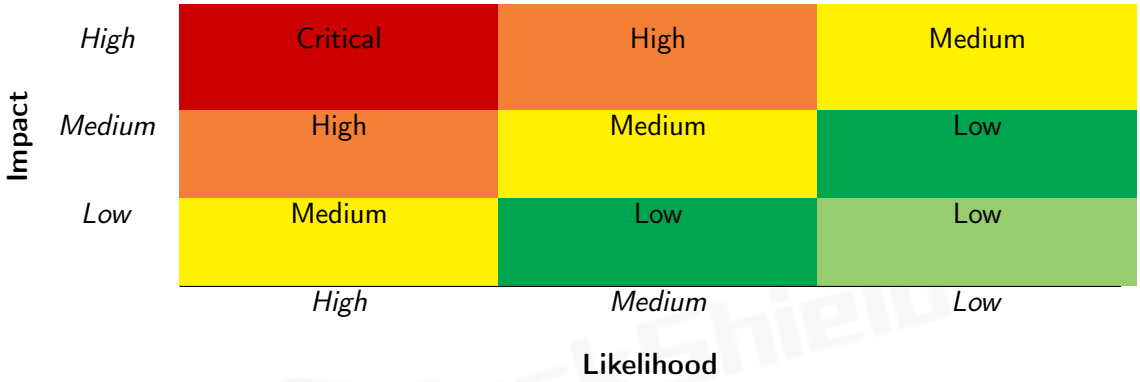
And this is the commit ID after all fixes for the issues found in the audit have been checked in.

- https://github.com/bartend3r/topshelf-contracts.git (059de14)

## 1.2 About PeckShield

PeckShield Inc. [13] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification




## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [12]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full Audit Checklist

| Category | Checklist Items |
|---|---|
| **Basic Coding Bugs** | Constructor Mismatch |
| | Ownership Takeover |
| | Redundant Fallback Function |
| | Overflows & Underflows |
| | Reentrancy |
| | Money-Giving Bug |
| | Blackhole |
| | Unauthorized Self-Destruct |
| | Revert DoS |
| | Unchecked External Call |
| | Gasless Send |
| | Send Instead Of Transfer |
| | Costly Loop |
| | (Unsafe) Use Of Untrusted Libraries |
| | (Unsafe) Use Of Predictable Variables |
| | Transaction Ordering Dependence |
| | Deprecated Uses |
| **Semantic Consistency Checks** | Semantic Consistency Checks |
| **Advanced DeFi Scrutiny** | Business Logics Review |
| | Functionality Checks |
| | Authentication Management |
| | Access Control & Authorization |
| | Oracle Security |
| | Digital Asset Escrow |
| | Kill-Switch Mechanism |
| | Operation Trails & Event Generation |
| | ERC20 Idiosyncrasies Handling |
| | Frontend-Contract Integration |
| | Deployment Consistency |
| | Holistic Risk Management |
| **Additional Recommendations** | Avoiding Use of Variadic Byte Array |
| | Using Fixed Compiler Version |
| | Making Visibility Level Explicit |
| | Making Type Inference Explicit |
| | Adhering To Function Declaration Strictly |
| | Following Other Best Practices |

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [11], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

## 1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

| Category | Summary |
|---|---|
| **Configuration** | Weaknesses in this category are typically introduced during the configuration of the software. |
| **Data Processing Issues** | Weaknesses in this category are typically found in functionality that processes data. |
| **Numeric Errors** | Weaknesses in this category are related to improper calculation or conversion of numbers. |
| **Security Features** | Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.) |
| **Time and State** | Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads. |
| **Error Conditions, Return Values, Status Codes** | Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function. |
| **Resource Management** | Weaknesses in this category are related to improper management of system resources. |
| **Behavioral Issues** | Weaknesses in this category are related to unexpected behaviors from code that an application uses. |
| **Business Logic** | Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application. |
| **Initialization and Cleanup** | Weaknesses in this category occur in behaviors that are used for initialization and breakdown. |
| **Arguments and Parameters** | Weaknesses in this category are related to improper use of arguments or parameters within function calls. |
| **Expression Issues** | Weaknesses in this category are related to incorrectly written expressions within code. |
| **Coding Practices** | Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained. |

# 2 | Findings

## 2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Topshelf` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

| Severity | # of Findings | |
|---|---|---|
| Critical | 0 | |
| High | 0 | |
| Medium | 2 | ■■ |
| Low | 4 | ■■■■ |
| Informational | 1 | ■ |
| Total | 7 | |

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

## 2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Topshelf Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Low | Improved Validation in LQTYToken/LUSDToken::permit() | Coding Practices | Resolved |
| PVE-002 | Medium | Possible Sandwich/MEV For Reduced Minted LPs) | Time And State | Resolved |
| PVE-003 | Low | Improved Status in PriceFeed::_fetchPrice() | Business Logic | Resolved |
| PVE-004 | Low | Accommodation of Non-ERC20-Compliant Tokens | Business Logic | Resolved |
| PVE-005 | Medium | Potential Overflow Mitigation in notifyRewardAmount() | Numeric Errors | Confirmed |
| PVE-006 | Low | Improved Trove Close Logic in TroveManager | Business Logic | Resolved |
| PVE-007 | Informational | Consistent Event Generation Of CollateralAddressChanged | Coding Practices | Confirmed |

Besides the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

# 3 Detailed Results

## 3.1 Improved Validation in LQTYToken/LUSDToken::permit()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LQTYToken`, `LUSDToken`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [3]

### Description

The `Topshelf` protocol has two tokens `LUSDToken` and `LIQRToken`, each supporting the `EIP2612` functionality. In particular, the `permit()` function is introduced to simplify the token transfer process.

To elaborate, we show below this helper routine from the `LIQRToken` contract. This routine ensures that the given `owner` is indeed the one who signs the approve request. Note that the internal implementation makes use of the `ecrecover()` precompile for validation. It comes to our attention that the precompile-based validation needs to properly ensure the signer, i.e., `owner`, is not equal to `address(0)`. This issue is also applicable to the `LUSDToken` token contract.

```
124     function permit
125     (
126         address owner,
127         address spender,
128         uint amount,
129         uint deadline,
130         uint8 v,
131         bytes32 r,
132         bytes32 s
133     )
134         external
135         override
136     {
137         require(deadline >= now, 'expired deadline');
138         bytes32 digest = keccak256(abi.encodePacked('\x19\x01',
139                          domainSeparator(), keccak256(abi.encode(
```

```
140                         _PERMIT_TYPEHASH, owner, spender, amount,
141                         _nonces[owner]++, deadline))));
142         address recoveredAddress = ecrecover(digest, v, r, s);
143         require(recoveredAddress == owner, 'invalid signature');
144         _approve(owner, spender, amount);
145     }
```

Listing 3.1: `LIQRToken::permit()`

**Recommendation** Strengthen the `permit()` routine to ensure the `owner` is not equal to `address(0)`.

**Status** The issue has been fixed by this commit: `2ec75f3`.

## 3.2 Possible Sandwich/MEV For Reduced Minted LPs

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `InitialLiquidityPool`
- Category: Time and State [9]
- CWE subcategory: CWE-682 [4]

### Description

The `Topshelf` protocol has an `InitialLiquidityPool` contract, which allows users to contribute the initial liquidity to reach the intended soft cap within the specified deposit period. Once the deposit period is finished and the soft cap has been reached, the contributed liquidity will be added and the rewards can be started to stream for contributors. While reviewing this liquidity-addition logic, we notice the current implementation may be improved.

To elaborate, we show below the related `addLiquidity()` routine. As the name indicates, it has a rather straightforward logic in adding the liquidity to the respective `UniswapV2Pair` with two constituent tokens: `WETH` and `rewardToken`.

```
221     // after the deposit period is finished and the soft cap has been reached,
222     // call this method to add liquidity and begin reward streaming for contributors
223     function addLiquidity() public onlyOwner {
224         require(!isKilled, "Killed");
225         require(block.timestamp >= depositEndTime, "Deposits are still open");
226         require(totalReceived >= softCapInETH, "Soft cap not reached");
227         require(!liquidityAdded, "Liquidity already added");
228         uint256 amount = address(this).balance;
229         WETH.deposit{ value: amount }();
230         WETH.transfer(lpToken, amount);
231         rewardToken.transfer(lpToken, rewardTokenLpAmount);
232         IUniswapV2Pair(lpToken).mint(treasury);
```

```
234         currentDepositTotal = totalReceived;
235         currentRewardTotal = rewardTokenSaleAmount;
236         liquidityAdded = true;
237     }
```

Listing 3.2: `InitialLiquidityPool::addLiquidity()`

We notice the current logic does not validate the amount of minted liquidity. In other words, the current approach does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller amount of minted LPs.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

**Recommendation** Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

**Status** The issue has been resolved as the team clarifies that the added liquidity is the first liquidity for the pool. And no sandwich attack is possible because prior to adding liquidity, there are no tokens in circulation.

## 3.3 Improved Oracle Status in PriceFeed::_fetchPrice()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PriceFeed`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [5]

### Description

The `Topshelf` protocol is unique in supporting dual oracles, which necessitate the examination of current oracle states. In total, there are five different oracle states, i.e., `chainlinkWorking`, `usingBandChainlinkUntrusted`, `bothOraclesUntrusted`, `usingBandChainlinkFrozen`, and `usingChainlinkBandUntrusted`. While examining possible transition from the fourth state, we notice the transition logic can be revisited.

To elaborate, we show below the code snippet from the `_fetchPrice()` function. This function is designed to fetch the current price and adjust the current oracle state accordingly. Starting from the fourth state `usingBandChainlinkFrozen`, the current logic considers the conditions of `!_chainlinkIsFrozen(chainlinkResponse)` (line 283) and `_bandIsBroken(bandResponse)` (line 304) to still yield `usingBandChainlinkFrozen` as the next state, which in fact can be better adjusted as `usingChainlinkBandFrozen`.

```
265         // --- CASE 4: Using Band, and Chainlink is frozen ---
266         if (status == Status.usingBandChainlinkFrozen) {
267             if (_chainlinkIsBroken(chainlinkResponse, prevChainlinkResponse)) {
268                 // If both Oracles are broken, return last good price
269                 if (_bandIsBroken(bandResponse)) {
270                     _changeStatus(Status.bothOraclesUntrusted);
271                     return lastGoodPrice;
272                 }
273
274                 // If Chainlink is broken, remember it and switch to using Band
275                 _changeStatus(Status.usingBandChainlinkUntrusted);
276
277                 if (_bandIsFrozen(bandResponse)) {return lastGoodPrice;}
278
279                 // If Band is working, return Band current price
280                 return _storeBandPrice(bandResponse);
281             }
282
283             if (_chainlinkIsFrozen(chainlinkResponse)) {
284                 // if Chainlink is frozen and Band is broken, remember Band broke, and
                        return last good price
285                 if (_bandIsBroken(bandResponse)) {
286                     _changeStatus(Status.usingChainlinkBandUntrusted);
287                     return lastGoodPrice;
288                 }
289
290                 // If both are frozen, just use lastGoodPrice
291                 if (_bandIsFrozen(bandResponse)) {return lastGoodPrice;}
292
293                 // if Chainlink is frozen and Band is working, keep using Band (no
                        status change)
294                 return _storeBandPrice(bandResponse);
295             }
296
297             // if Chainlink is live and Band is broken, remember Band broke, and return
                    Chainlink price
298             if (_bandIsBroken(bandResponse)) {
299                 _changeStatus(Status.usingChainlinkBandUntrusted);
300                 return _storeChainlinkPrice(chainlinkResponse);
301             }
302
303              // If Chainlink is live and Band is frozen, just use last good price (no
                     status change) since we have no basis for comparison
304             if (_bandIsFrozen(bandResponse)) {return lastGoodPrice;}
```

```
305
306             // If Chainlink is live and Band is working, compare prices. Switch to
                    Chainlink
307             // if prices are within 5%, and return Chainlink price.
308             if (_bothOraclesSimilarPrice(chainlinkResponse, bandResponse)) {
309                 _changeStatus(Status.chainlinkWorking);
310                 return _storeChainlinkPrice(chainlinkResponse);
311             }
312
313             // Otherwise if Chainlink is live but price not within 5% of Band, distrust
                    Chainlink, and return Band price
314             _changeStatus(Status.usingBandChainlinkUntrusted);
315             return _storeBandPrice(bandResponse);
316         }
```

Listing 3.3: `PriceFeed::_fetchPrice()`

**Recommendation** Apply the proper state-transition logic in `_fetchPrice()` as elaborated earlier.

**Status** The issue has been fixed by this commit: `6f75a88`.

## 3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [6]

### Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
64      function transfer(address _to, uint _value) returns (bool) {
65          //Default assumes totalSupply can't be over max (2^256 - 1).
66          if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
67              balances[msg.sender] -= _value;
68              balances[_to] += _value;
69              Transfer(msg.sender, _to, _value);
70              return true;
71          } else { return false; }
72      }
73
74      function transferFrom(address _from, address _to, uint _value) returns (bool) {
75          if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
                balances[_to] + _value >= balances[_to]) {
76              balances[_to] += _value;
77              balances[_from] -= _value;
78              allowed[_from][msg.sender] -= _value;
79              Transfer(_from, _to, _value);
80              return true;
81          } else { return false; }
82      }
```

Listing 3.4: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `claimReward()` routine in the `InitialLiquidityPool` contract. If the `USDT` token is supported as `rewardToken`, the unsafe version of `rewardToken.transfer(msg.sender, amount)` (line 278) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
270     // claim a pending `rewardToken` balance
271     function claimReward() external {
272         require(!isKilled, "Killed");
273         require(liquidityAdded, "Liquidity was not added");
274         uint256 amount = claimable(msg.sender);
275         userAmounts[msg.sender].streamed = userAmounts[msg.sender].streamed.add(
276             amount
277         );
278         rewardToken.transfer(msg.sender, amount);
279         emit Claim(msg.sender, amount);
280     }
```

Listing 3.5: `InitialLiquidityPool::claimReward()`

Note this issue is also applicable to other routines, including `InitialLiquidityPool::earlyExit()/addLiquidity()`, `CollSurplusPool::claimColl()`, and `FlashLender::flashLoan()`. For the `safeApprove()`

support, there is a need to approve twice: the first time resets the allowance to zero and the second time approves the intended amount. And this affects other routines, including `LQTYTreasury::setAddresses()`.

**Recommendation** Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`/`transfer()`/`transferFrom()`.

**Status** The issue has been resolved as the protocol only intends to interact with tokens that adhere to the most common ERC20 behavior, which returns true if not reverted.

## 3.5 Potential Overflow Mitigation in notifyRewardAmount()

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `MultiRewards`
- Category: Numeric Errors [10]
- CWE subcategory: CWE-190 [2]

### Description

The `Topshelf` protocol has a built-in incentivizer mechanism. In this section, we focus on a routine, i.e., `rewardPerToken()`, which is responsible for calculating the reward rate for each staked token and it is part of the `updateReward` modifier that would be invoked up-front for almost every public function in `MultiRewards` to update and use the latest reward rate.

The reason is due to the known potential overflow pitfall when a new oversized reward amount is added into the pool. In particular, as the `rewardPerToken()` routine involves the multiplication of three `uint256` integer, it is possible for their multiplication to have an undesirable overflow (line 82), especially when the `rewardRate` is largely controlled through the `notifyRewardAmount()` function.

```
161     modifier updateReward(address account) {
162         for (uint i; i < rewardTokens.length; i++) {
163             address token = rewardTokens[i];
164             rewardData[token].rewardPerTokenStored = rewardPerToken(token);
165             rewardData[token].lastUpdateTime = lastTimeRewardApplicable(token);
166             if (account != address(0)) {
167                 rewards[account][token] = earned(account, token);
168                 userRewardPerTokenPaid[account][token] = rewardData[token].
                        rewardPerTokenStored;
169             }
170         }
171         _;
172     }
```

Listing 3.6: `MultiRewards::updateReward()`

```
72      function lastTimeRewardApplicable(address _rewardsToken) public view returns (
            uint256) {
73          return Math.min(block.timestamp, rewardData[_rewardsToken].periodFinish);
74      }
75
76      function rewardPerToken(address _rewardsToken) public view returns (uint256) {
77          if (_totalSupply == 0) {
78              return rewardData[_rewardsToken].rewardPerTokenStored;
79          }
80          return
81              rewardData[_rewardsToken].rewardPerTokenStored.add(
82                  lastTimeRewardApplicable(_rewardsToken).sub(rewardData[_rewardsToken].
                        lastUpdateTime).mul(rewardData[_rewardsToken].rewardRate).mul(1e18).
                        div(_totalSupply)
83              );
84      }
```

Listing 3.7: `MultiRewards::rewardPerToken()`

Apparently, this issue is made possible if the reward amount is given as the argument to `notifyRewardAmount()` such that the calculation of `rewardData[_rewardsToken].rewardRate.mul(1e18)` always overflows, hence locking all deposited funds! Note that an authentication check on the caller of `notifyRewardAmount()` greatly alleviates such concern. Currently, only the intended rewards distributor is able to call `notifyRewardAmount()` and this address is set by the owner. Apparently, if the owner is a normal address, it may put users' funds at risk. To mitigate this issue, it is necessary to have the ownership under the governance control and ensure the given reward amount will not be oversized to overflow and lock users' funds.

**Recommendation** Mitigate the potential overflow risk in the incentivize pool. An example revision is shown below.

```
135     function notifyRewardAmount(address _rewardsToken, uint256 reward) external
            updateReward(address(0)) {
136         require(rewardData[_rewardsToken].rewardsDistributor[msg.sender], "Invalid
                caller");
137         require(reward < uint256(-1) / 10**22, "rewards too large, would lock");
138
139         if (block.timestamp >= rewardData[_rewardsToken].periodFinish) {
140             rewardData[_rewardsToken].rewardRate = reward.div(rewardsDuration);
141         } else {
142             uint256 remaining = rewardData[_rewardsToken].periodFinish.sub(block.
                    timestamp);
143             uint256 leftover = remaining.mul(rewardData[_rewardsToken].rewardRate);
144             rewardData[_rewardsToken].rewardRate = reward.add(leftover).div(
                    rewardsDuration);
145         }
146
147         rewardData[_rewardsToken].lastUpdateTime = block.timestamp;
148         rewardData[_rewardsToken].periodFinish = block.timestamp.add(rewardsDuration);
149         emit RewardAdded(_rewardsToken, reward);
```

```
150     }
```

Listing 3.8: Revised MultiRewards::notifyRewardAmount()

**Status** This issue has been confirmed.

## 3.6 Improved Trove Close Logic in TroveManager

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TroveManager`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [6]

### Description

At the core of `Topshelf` is the `TroveManager` contract which contains the logic to open, adjust and close various troves. Note each trove is in essence an individual collateralized debt position for borrowing users. While reviewing the current trove-closing logic, we notice the current implementation can be improved.

To elaborate, we show below the related `_closeTrove()` routine. The current logic properly releases unused states, including the trove `coll`, `debt`, as well as the associated `rewardSnapshots`. However, it does not release the trove index in the global owners, i.e., `TroveOwners`. The release of `arrayIndex` needs to be performed after the call `_removeTroveOwner()` is completed.

```
1249    function _closeTrove(address _borrower, Status closedStatus) internal {
1250        assert(closedStatus != Status.nonExistent && closedStatus != Status.active);
1251
1252        uint TroveOwnersArrayLength = TroveOwners.length;
1253        _requireMoreThanOneTroveInSystem(TroveOwnersArrayLength);
1254
1255        Troves[_borrower].status = closedStatus;
1256        Troves[_borrower].coll = 0;
1257        Troves[_borrower].debt = 0;
1258
1259        rewardSnapshots[_borrower].ETH = 0;
1260        rewardSnapshots[_borrower].LUSDDebt = 0;
1261
1262        _removeTroveOwner(_borrower, TroveOwnersArrayLength);
1263        sortedTroves.remove(_borrower);
1264    }
```

Listing 3.9: `TroveManager::_closeTrove()`

**Recommendation** Release all unused states once a trove is closed. An example revision is shown below:

```
1249    function _closeTrove(address _borrower, Status closedStatus) internal {
1250        assert(closedStatus != Status.nonExistent && closedStatus != Status.active);
1251
1252        uint TroveOwnersArrayLength = TroveOwners.length;
1253        _requireMoreThanOneTroveInSystem(TroveOwnersArrayLength);
1254
1255        Troves[_borrower].status = closedStatus;
1256        Troves[_borrower].coll = 0;
1257        Troves[_borrower].debt = 0;
1258
1259        rewardSnapshots[_borrower].ETH = 0;
1260        rewardSnapshots[_borrower].LUSDDebt = 0;
1261
1262        _removeTroveOwner(_borrower, TroveOwnersArrayLength);
1263        sortedTroves.remove(_borrower);
1264        Troves[_borrower].arrayIndex = 0;
1265    }
```

Listing 3.10: `TroveManager::_closeTrove()`

**Status** The issue has been fixed by this commit: `1cf6c9d`.

## 3.7 Consistent Event Generation Of CollateralAddressChanged

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

### Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `CollSurplusPool` contract as an example. This contract has a privileged public function to configure the current contract addresses after their deployment. While examining the events that reflect their update, we notice there is a lack of emitting a related event to reflect the `collateralToken` update. It comes to our attention that the same routine in `BorrowerOperations` has properly emitted the respective event `CollateralAddressChanged`.

```
490     function setAddresses(
491         address _borrowerOperationsAddress,
492         address _troveManagerAddress,
493         address _activePoolAddress,
494         address _collateralTokenAddress
495     )
496         external
497         override
498         onlyOwner
499     {
500         checkContract(_borrowerOperationsAddress);
501         checkContract(_troveManagerAddress);
502         checkContract(_activePoolAddress);
503
504         borrowerOperationsAddress = _borrowerOperationsAddress;
505         troveManagerAddress = _troveManagerAddress;
506         activePoolAddress = _activePoolAddress;
507         collateralToken = IERC20(_collateralTokenAddress);
508
509         emit BorrowerOperationsAddressChanged(_borrowerOperationsAddress);
510         emit TroveManagerAddressChanged(_troveManagerAddress);
511         emit ActivePoolAddressChanged(_activePoolAddress);
512
513         _renounceOwnership();
514     }
```

Listing 3.11: `CollSurplusPool::setAddresses()`

**Recommendation** Properly emit respective events when a new `collateralToken` becomes effective. This affects a number of contracts, including `ActivePool`, `CollSurplusPool`, `DefaultPool`, and `StabilityPool`.

**Status** This issue has been confirmed.

# 4 Conclusion

In this audit, we have analyzed the design and implementation of the Topshelf protocol, which is a decentralized borrowing protocol that allows to draw low-interest loans against a variety of tokens used as collateral. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

# References


[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[5] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[6] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[7] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[8] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[9] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[10] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[11] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[12] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[13] PeckShield. PeckShield Inc. https://www.peckshield.com.