SMART CONTRACT AUDIT REPORT

for

Torches Protocol

Prepared By: Xiaomi Huang

PeckShield

June 25, 2022

Document Properties

Client	Torches Finance
Title	Smart Contract Audit Report
Target	Torches
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 25, 2022	Xuxian Jiang	Final Release
1.0-rc1	June 5, 2022	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Torches` protocol, 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 audited protocol 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 Torches

`Torches` is a lending and borrowing protocol with the goal of developing a cross-chain money market. The protocol designs are architected and inspired based on `Compound` to allow users to utilize their cryptocurrencies by supplying collateral to the protocol. The supplied collateral enables the user to borrow supported assets while maintaining an over-collateralized borrow position. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Torches

Item	Description
Name	Torches Finance
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 25, 2022

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

- https://github.com/TorchesFinance/torches-protocol.git (21dfdef)

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

- https://github.com/TorchesFinance/torches-protocol.git (TBD)

1.2 About PeckShield

PeckShield Inc. [11] 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

Impact \ Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- 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.

To evaluate the risk, we go through a list of check 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

Table 1.3: The Full List of Check Items

Category	Check Item
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

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) [9], 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.

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 Logics	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 `Torches` implementation. 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	1	■
Medium	1	■
Low	3	■ ■ ■
Informational	0	
Total	5	

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 that 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 1 high-severity vulnerability, 1 medium-severity vulnerability, and 3 low-severity vulnerabilities.

Table 2.1: Key Torches Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Uninitialized State Index DoS From Reward Activation	Business Logic	Resolved
PVE-002	Low	Interface Inconsistency Between CEther And CErc20	Coding Practices	Resolved
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Confirmed
PVE-004	Low	Non ERC20-Compliance Of CToken	Coding Practices	Resolved
PVE-005	Medium	Trust on Admin Keys	Security Features	Mitigated

Beside 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 Uninitialized State Index DoS From Reward Activation

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Comptroller`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `Torches` protocol provides incentive mechanisms that reward the protocol users. Specifically, the reward mechanism follows the same approach as the `COMP` reward in `Compound`. Our analysis on the related `COMP` reward in `Torches` shows the current logic needs to be improved.

To elaborate, we show below the initial logic of `_setCompSpeeds()` that kicks off the actual minting of protocol tokens. It comes to our attention that the initial supply-side index is configured on the conditions of `compSupplyState[cToken].index == 0` and `compSupplyState[cToken].block == 0` (line 901). However, for an already listed market with a current speed of 0, the first condition is indeed met while the second condition does not! The reason is that both supply-side state and borrow-side state have the associated block information updated, which is diligently performed via other helper pairs `updateCompSupplyIndex()/updateCompBorrowIndex()`. As a result, the `_setCompSpeedInternal()` logic does not properly set up the default supply-side index and the default borrow-side index.

```
    function _setCompSpeeds(address[] memory _allMarkets, uint[] memory _compSpeeds)
        public {
        // Check caller is admin
        require(msg.sender == admin);

        require(_allMarkets.length == _compSpeeds.length);

        for (uint i = 0; i < _allMarkets.length; i++) {
            _setCompSpeedInternal(_allMarkets[i], _compSpeeds[i]);
        }
    }
```

```
    function _setCompSpeedInternal(address _cToken, uint _compSpeed) internal {
        Market storage market = markets[_cToken];
        if (market.isComped == false) {
            _addCompMarketInternal(_cToken);
        }
        uint currentCompSpeed = compSpeeds[_cToken];
        uint currentSupplySpeed = currentCompSpeed >> 128;
        uint currentBorrowSpeed = uint128(currentCompSpeed);

        uint newSupplySpeed = _compSpeed >> 128;
        uint newBorrowSpeed = uint128(_compSpeed);
        if (currentSupplySpeed != newSupplySpeed) {
            updateCompSupplyIndex(_cToken);
        }
        if (currentBorrowSpeed != newBorrowSpeed) {
            Exp memory borrowIndex = Exp({mantissa: CToken(_cToken).borrowIndex()});
            updateCompBorrowIndex(_cToken, borrowIndex);
        }
        compSpeeds[_cToken] = _compSpeed;
    }
```

Listing 3.1: `Torchestroller::_setCompSpeeds()`

```
    function _addCompMarketInternal(address cToken) internal {
        Market storage market = markets[cToken];
        require(market.isListed == true, "!listed");
        require(market.isComped == false, "already added");

        market.isComped = true;
        emit MarketComped(CToken(cToken), true);

        if (compSupplyState[cToken].index == 0 && compSupplyState[cToken].block == 0) {
            compSupplyState[cToken] = CompMarketState({
                index: compInitialIndex,
                block: safe32(getBlockNumber(), ">32 bits")
            });
        }

        if (compBorrowState[cToken].index == 0 && compBorrowState[cToken].block == 0) {
            compBorrowState[cToken] = CompMarketState({
                index: compInitialIndex,
                block: safe32(getBlockNumber(), ">32 bits")
            });
        }
    }
```

Listing 3.2: `Comptroller::_addCompMarketInternal()`

When the reward speed is configured, since the supply-side and borrow-side state indexes are not initialized, any normal functionality such as `mint()` will be immediately reverted! This revert occurs inside the `distributeSupplierComp()/distributeBorrowerComp()` functions. Using the `distributeSupplierComp`

`()` function as an example, the revert is caused from the arithmetic operation `sub_(supplyIndex, supplierIndex)` (line 813). Since the `supplyIndex` is not properly initialized, it will be updated to a smaller number from an earlier invocation of `updateCompSupplyIndex()` (line 211). However, when the `distributeSupplierComp()` function is invoked, the `supplierIndex` is reset with `compInitialIndex` (line 810), which unfortunately reverts the arithmetic operation `sub_(supplyIndex, supplierIndex)`!

```
803     function distributeSupplierComp(address cToken, address supplier, bool distributeAll
            ) internal {
804         CompMarketState storage supplyState = compSupplyState[cToken];
805         Double memory supplyIndex = Double({mantissa: supplyState.index});
806         Double memory supplierIndex = Double({mantissa: compSupplierIndex[cToken][
                supplier]});
807         compSupplierIndex[cToken][supplier] = supplyIndex.mantissa;

809         if (supplierIndex.mantissa == 0 && supplyIndex.mantissa > 0) {
810             supplierIndex.mantissa = compInitialIndex;
811         }

813         Double memory deltaIndex = sub_(supplyIndex, supplierIndex);
814         uint supplierTokens = CToken(cToken).balanceOf(supplier);
815         uint supplierDelta = mul_(supplierTokens, deltaIndex);
816         uint supplierAccrued = add_(compAccrued[supplier], supplierDelta);
817         compAccrued[supplier] = transferComp(supplier, supplierAccrued, distributeAll ?
                0 : compClaimThreshold);
818         emit DistributedSupplierComp(CToken(cToken), supplier, supplierDelta,
                supplyIndex.mantissa);
819     }
```

Listing 3.3: `Comptroller::distributeSupplierComp()`

Recommendation Properly initialize the reward state indexes in the above affected `_addCompMarketInternal()` function. An example revision is shown as follows:

```
    function _addCompMarketInternal(address cToken) internal {
        Market storage market = markets[cToken];
        require(market.isListed == true, "!listed");
        require(market.isComped == false, "already added");

        market.isComped = true;
        emit MarketComped(CToken(cToken), true);

        if (compSupplyState[cToken].index == 0) {
            compSupplyState[cToken].index = compInitialIndex;
        }
        compSupplyState[cToken].block = safe32(getBlockNumber();

        if (compBorrowState[cToken].index == 0) {
            compBorrowState[cToken].index = compInitialIndex;
        }
        compBorrowState[cToken].block = safe32(getBlockNumber();
```

```
}
```

Listing 3.4: `Comptroller::_addCompMarketInternal()`

Status The issue has been resolved as the team confirms the proper validation fo the related configuration.

3.2 Interface Inconsistency Between CEther And CErc20

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned in Section 3.1, each asset supported by the `Torches` protocol is integrated through a so-called `CToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. And `CTokens` are the primary means of interacting with the `Torches` protocol when a user wants to `mint()`, `redeem()`, `borrow()`, `repay()`, `liquidate()`, or `transfer()`. Moreover, there are currently two types of `CTokens`: `CErc20` and `CEther`. Both types expose the ERC20 interface and they wrap an underlying `ERC20` asset and `Ether`, respectively.

While examining these two types, we notice their interfaces are surprisingly different. Using the `replayBorrow()` function as an example, the `CErc20` type returns an error code while the `CEther` type simply reverts upon any failure. The similar inconsistency is also present in other routines, including `repayBorrowBehalf()`, `mint()`, and `liquidateBorrow()`.

```
/**
 * @notice Sender repays their own borrow
 * @param repayAmount The amount to repay
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function repayBorrow(uint repayAmount) external returns (uint) {
    (uint err,) = repayBorrowInternal(repayAmount);
    return err;
}
```

Listing 3.5: CErc20::repayBorrow()

```
/**
 * @notice Sender repays their own borrow
 * @dev Reverts upon any failure
 */
```

```
    function repayBorrow() external payable {
        (uint err,) = repayBorrowInternal(msg.value);
        requireNoError(err, "repayBorrow failed");
    }
```

Listing 3.6: CEther::repayBorrow()

It is also worth mentioning that the `CErc20` type supports `_addReserves` while the `CEther` type does not.

Recommendation Ensure the consistency between these two types: `CErc20` and `CEther`.

Status This issue has been confirmed. Considering that this is part of the original `Compound` code base, the team decides to leave it as is to minimize the difference from the original `Compound` and reduce the risk of introducing bugs as a result of changing the behavior.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Reservoir`
- Category: Coding Practices [7]
- CWE subcategory: CWE-628 [4]

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."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
```

```
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.7: 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 `drip()` routine in the `Reservoir` contract. If the `USDT` token is supported as `token`, the unsafe version of `token_.transfer(target_, toDrip_)` (line 63) 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)!

```
45  function drip() public returns (uint) {
46    // First, read storage into memory
47    EIP20Interface token_ = token;
48    uint reservoirBalance_ = token_.balanceOf(address(this)); // TODO: Verify this is a static call
49    uint dripRate_ = dripRate;
50    uint dripStart_ = dripStart;
51    uint dripped_ = dripped;
52    address target_ = target;
53    uint blockNumber_ = block.number;
54
55    // Next, calculate intermediate values
56    uint dripTotal_ = mul(dripRate_, blockNumber_ - dripStart_, "dripTotal overflow");
57    uint deltaDrip_ = sub(dripTotal_, dripped_, "deltaDrip underflow");
58    uint toDrip_ = min(reservoirBalance_, deltaDrip_);
59    uint drippedNext_ = add(dripped_, toDrip_, "tautological");
60
61    // Finally, write new `dripped` value and transfer tokens to target
62    dripped = drippedNext_;
63    token_.transfer(target_, toDrip_);
64
65    return toDrip_;
```

```
}
```

Listing 3.8: Reservoir::drip()

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

Status The issue has been confirmed.

3.4 Non ERC20-Compliance Of CToken

- ID: PVE-004
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: CToken
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

Description

Each asset supported by the Torches protocol is integrated through a so-called CToken contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting CToken, users can earn interest through the CToken's exchange rate, which increases in value relative to the underlying asset, and further gains the ability to use CToken as collateral. In the following, we examine the ERC20 compliance of these CToken.

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Our analysis shows that there are several ERC20 inconsistency or incompatibility issues found in the CToken contract. Specifically, the current transfer() function simply returns the related error code if the sender does not have sufficient balance to spend. A similar issue is also present in the transferFrom() function that does not revert when the sender does not have the sufficient balance or the message sender does not have the enough allowance.

In the surrounding two tables, we outline the respective list of basic view-only functions (Table 3.1) and key state-changing functions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g.,

Table 3.1: Basic View-Only Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Recommendation Revise the `CToken` implementation to ensure its ERC20-compliance.

Status The issue has been confirmed.

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	×
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

Table 3.3: Additional `Opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `Torches` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and marketing adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function _setCreditLimit(address protocol, uint creditLimit) public onlyOwner {
        require(isContract(protocol), "contract required");
        require(creditLimits[protocol] != creditLimit, "no change");

        creditLimits[protocol] = creditLimit;
        emit CreditLimitChanged(protocol, creditLimit);
    }

    function _setCompToken(address _compToken) public onlyOwner {
        address oldCompToken = compToken;
        compToken = _compToken;
        emit NewCompToken(oldCompToken, compToken);
```

```
    }

    function _setSafetyVault(address _safetyVault) public onlyOwner {
        address oldSafetyVault = safetyVault;
        safetyVault = _safetyVault;
        emit NewSafetyVault(oldSafetyVault, safetyVault);
    }

    function _setSafetyVaultRatio(uint _safetyVaultRatio) public onlySafetyGuardian {
        require(_safetyVaultRatio < 1e18, "!safetyVaultRatio");

        uint oldSafetyVaultRatio = safetyVaultRatio;
        safetyVaultRatio = _safetyVaultRatio;
        emit NewSafetyVaultRatio(oldSafetyVaultRatio, safetyVaultRatio);
    }
```

Listing 3.9: Example `setters` in the `TorchesConfig` Contract

Apparently, if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

```
contract TransparentUpgradeableProxyImpl is TransparentUpgradeableProxy {
  constructor(
    address _logic,
    address _admin,
    bytes memory _data
  ) public payable TransparentUpgradeableProxy(_logic, _admin, _data) {}
}
```

Listing 3.10: `TransparentUpgradeableProxyImpl::constructor()`

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed with the team. For the time being, the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Torches` protocol, which is a lending and borrowing protocol with the goal of developing a cross-chain money market. The protocol designs are architected and inspired based on `Compound` to allow users to utilize their cryptocurrencies by supplying collateral to the protocol. The supplied collateral enables the user to borrow supported assets while maintaining an over-collateralized borrow position. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

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

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.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 VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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