SMART CONTRACT AUDIT REPORT

for

MATRIXPORT

Prepared By: Yiqun Chen

Hangzhou, China
July 7, 2021

Document Properties

Client	Matrixport
Title	Smart Contract Audit Report
Target	ccTokens
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Jeff Liu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0.1	July 7, 2021	Xuxian Jiang	Final Release Amended
1.0	November 22, 2020	Xuxian Jiang	Final Release
1.0-rc	November 22, 2020	Xuxian Jiang	Release Candidate
0.2	November 20, 2020	Xuxian Jiang	Additional Findings
0.1	November 18, 2020	Xuxian Jiang	Initial Draft

Contact

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

Name	Yiqun Chen
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 **ccTokens**, we outline in this 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 contract can be further improved due to the presence of several issues. This document outlines our audit results.

1.1 About ccTokens

`ccTokens` are ERC20 tokens that project blockchain assets such as BTC in Ethereum on a 1:1 basis. It enables seamless integration of each crypto asset into the Ethereum ecosystem. All ccTokens reserves are safeguarded by qualified third-party custodians. Meanwhile, a multi-signature mechanism is adopted for its crucial aspects such as mining and burning, allowing on-the-chain verification. Therefore, it provides cross-chain asset services that are both transparent and reliable. At the same time, its blacklist mechanism maximizes security and its applications.

The basic information of ccTokens is as follows:

Table 1.1: Basic Information of ccTokens

Item	Description
Issuer	Matrixport
Website	https://www.crosschain.network/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 7, 2021

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

- https://github.com/ccTokens/Smart-Contract.git (466ffc6)

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

- https://github.com/ccTokens/Smart-Contract.git (5c490c2)

1.2 About PeckShield

PeckShield Inc. [18] 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 [13]:

- 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 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

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 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) [12], 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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 ccTokens 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 logics, 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	1	■
Low	2	■ ■
Informational	3	■ ■ ■
Total	6	

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 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 3 informational recommendations.

Table 2.1: Key ccTokens Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved transferFrom() in ERC20Basic	Business Logic	Fixed
PVE-002	Informational	Removal of Unused Code	Coding Practices	Fixed
PVE-003	Informational	Suggested Adherence of Checks-Effects-Interactions	Time and State	Fixed
PVE-004	Low	Improved Validity Checks in removeOwner()	Error Conditions, Return Values, Status Codes	Fixed
PVE-005	Informational	Suggested transactionExists() in revokeConfirmation()/executeTransaction()	Error Conditions, Return Values, Status Codes	Fixed
PVE-006	Medium	Trust Issue of Admin Keys Behind Custodian	Security Features	Fixed

Please refer to Section 3 for details.

3 Detailed Results

3.1 Improved transferFrom() in ERC20Basic

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ERC20Basic
- Category: Business Logic [9]
- CWE subcategory: CWE-754 [6]

Description

ccTokens are ERC20-compliant tokens that project blockchain assets such as BTC in Ethereum on a 1:1 basis. Accordingly, there is a need for their contract implementation, i.e., ERC20Basic, to follow the ERC20 specification. As the first step 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.

Our analysis shows that there is no ERC20 inconsistency or incompatibility issue found in the audited ccTokens. In the following 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.

Meanwhile, we notice in the transferFrom() routine, there is a common practice that is missing but widely used in other ERC20 contracts. Specifically, when msg.sender = _from, the current transferFrom() implementation disallows the token transfer if msg.sender has not explicitly allows spending from herself yet. A common practice will whitelist this special case and allow transferFrom() if msg.sender = _from even there is no allowance specified.

```
function transferFrom(
    address _from,
    address _to,
    uint256 _value
)
override public notPaused notBlocked
```

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	✓

```
    returns (bool)
    {
        require(_notBlocked(_from), "from-address has been blocked");
        require(_notBlocked(_to), "to-address has been blocked");
        require(_value <= balances[_from], "insufficient balance");
        require(_value <= allowed[_from][msg.sender], "value > allowed");
        require(_to != address(0), "invalid to-address");

        balances[_from] = balances[_from].sub(_value);
        balances[_to] = balances[_to].add(_value);
        allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
        emit Transfer(_from, _to, _value);
        return true;
    }
```

Listing 3.1: ERC20Basic.sol

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., ERC777), 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 Improve the transferFrom() logic by considering the special case when

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	✓
Approve() 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	—
Pausible	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	✓
Hookable	The token contract allows the sender/recipient to be notified while sending/receiving tokens	—
Permittable	The token contract allows for unambiguous expression of an intended spender with the specified allowance in an off-chain manner (e.g., a permit() call to properly set up the allowance with a signature).	—

msg.sender = _from. In the meantime, consider the support of permit() (in EIP-2612) for better integration and usability.

Status This issue has been fixed in the commit: 5c490c2.

3.2 Removal of Unused Code

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: MemberMgr
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

ccTokens makes good use of a number of reference contracts, such as ERC20Basic, NamedERC20, Ownable , and SafeMath to facilitate its code implementation and organization. For example, the ccToken smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `MemberMgr` contract, there is an enumerable named `MerchantStatus` with three states: `STOPPED`, `VALID`, `REMOVED`. The first two states are indeed necessary, but not the third `REMOVED` state. This unused state is apparently left from an already deprecated function. With that, we can simply drop the `REMOVED` state.

```
45     function requireMerchant(address _who) override public view {
46         MerchantStatusData memory merchantState = merchantStatus[_who];
47         if (!merchantState._exist) {
48             require(false, "not a merchant");
49             assert(false);
50         }
51 
52         if (merchantState.status == MerchantStatus.STOPPED) {
53             require(false, "merchant has been stopped");
54             assert(false);
55         }
56 
57         require(merchantState.status == MerchantStatus.VALID, "merchant not valid");
58     }
```

Listing 3.2: MemberMgr.sol

In addition, we notice the `requireMerchant()` routine can be revised to remove unreachable code in its implementation. In particular, the two `asset` statements (lines 49 and 54) are not used and can be safely removed. Also, the respective `require(false)` statements can be effectively combined with conditional checks for better readability and improved conciseness.

Recommendation Remove unreachable code in `MemberMgr` and revise the `requireMerchant()` routine as follows:

```
    function requireMerchant(address _who) override public view {
        MerchantStatusData memory merchantState = merchantStatus[_who];
        require(merchantState._exist, "not a merchant");
        require(merchantState.status != MerchantStatus.STOPPED, "merchant has been
            stopped");
        require(merchantState.status == MerchantStatus.VALID, "merchant not valid");
    }
```

Listing 3.3: MemberMgr.sol (revised)

Status This issue has been fixed in the commit: 5c490c2.

3.3 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `MintFactory`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [5]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [21] exploit, and the recent `Uniswap/Lendf.Me` hack [19].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Specifically, in the `MintFactory` contract, the `confirmMintRequest()` function (see the code snippet below) is provided to externally call a `controller` contract to mint `ccTokens`. Though this `controller` contract is trusted, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 292) starts before effecting the update on internal states (lines 293 – 295), hence violating the principle. In this particular case, if the external contract has some hidden logic that may be capable of launching `re-entrancy` via the very same `confirmMintRequest()` function.

```
285     function confirmMintRequest(bytes32 requestHash, uint amount) external onlyCustodian
            returns (bool) {
286         uint blockNo = block.number;
287         Request memory request = getPendingMintRequest(requestHash);
288         require(blockNo > request.requestBlockNo, "confirmMintRequest failed");
289
290         require(blockNo - 20 >= request.requestBlockNo, "confirmMintRequest failed, wait
                for 20 blocks");
291         uint seq = request.seq;
292         require(controller.mint(request.requester, amount), "mint failed");
293         mintRequests[seq].status = RequestStatus.APPROVED;
294         mintRequests[seq].amount = amount;
295         mintRequests[seq].confirmedBlockNo = blockNo;
296
297         emit MintConfirmed(
298             request.seq,
299             request.requester,
300             amount,
```

```
            request.btcAddress,
            request.btcTxId,
            blockNo,
            calcRequestHash(request)
        );
        return true;
    }
```

Listing 3.4: MintFactory.sol

Again, we need to emphasize that the `controller` contract is trusted and will not bring any security risk in current implementation.

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` best practice. The above routine can be revised as follows:

```
    function confirmMintRequest(bytes32 requestHash, uint amount) external onlyCustodian
         returns (bool) {
        uint blockNo = block.number;
        Request memory request = getPendingMintRequest(requestHash);
        require(blockNo > request.requestBlockNo, "confirmMintRequest failed");

        require(blockNo - 20 >= request.requestBlockNo, "confirmMintRequest failed, wait
             for 20 blocks");
        uint seq = request.seq;
        mintRequests[seq].status = RequestStatus.APPROVED;
        mintRequests[seq].amount = amount;
        mintRequests[seq].confirmedBlockNo = blockNo;

        require(controller.mint(request.requester, amount), "mint failed");
        emit MintConfirmed(
            request.seq,
            request.requester,
            amount,
            request.btcAddress,
            request.btcTxId,
            blockNo,
            calcRequestHash(request)
        );
        return true;
    }
```

Listing 3.5: MintFactory.sol

Status This issue has been fixed in the commit: 5c490c2.

3.4 Improved Validity Checks in removeOwner()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MultiSigWallet`
- Category: Status Codes [11]
- CWE subcategory: CWE-391 [3]

Description

ccTokens makes use of multi-sig wallets to mitigate possible risk from single centralized party. For example, both `MemberMgr` and `ccTokenController` have a privileged `owner` that is controlled by `MultiSigWallet` instances.

While reviewing the `MultiSigWallet` implementation, we notice the presence of `removeOwner()`, which is designed to remove an existing owner. In the following, we show its code snippet of current implementation. Our analysis shows that this routine has a corner case that can be better handled. Specifically, when the given owner for removal is the only remaining one or is the last one in the `owners` array, this routine fails to properly remove it from the array, even though the state has been properly marked as `false` (line 146).

```
139     /// @dev Allows to remove an owner. Transaction has to be sent by wallet.
140     /// @param owner Address of owner.
141     function removeOwner(address owner)
142     public
143     onlyWallet
144     ownerExists(owner)
145     {
146         isOwner[owner] = false;
147         for (uint i = 0; i < owners.length - 1; i++)
148             if (owners[i] == owner) {
149                 owners[i] = owners[owners.length - 1];
150                 owners.pop();
151                 break;
152             }
153         //        owners.length -= 1;

155         if (required > owners.length)
156             changeRequirement(owners.length);
157         emit OwnerRemoval(owner);
158     }
```

Listing 3.6: MultiSigWallet.sol

Because of the above corner case, it is possible to lead to an unexpected scenario where the to-be-removed owner still remains in the `owners` array without reducing the array length, hence giving an inaccurate safeguarding of the `required` threshold. In particular, we need to ensure there is always at least an owner to fulfill the duties.

Recommendation Properly handle the corner case and guarantee the presence of at least one owner as follows:

```
    /// @dev Allows to remove an owner. Transaction has to be sent by wallet.
    /// @param owner Address of owner.
    function removeOwner(address owner)
    public
    onlyWallet
    ownerExists(owner)
    {
        isOwner[owner] = false;
        for (uint i = 0; i < owners.length; i++)
            if (owners[i] == owner) {
                owners[i] = owners[owners.length - 1];
                owners.pop();
                break;
            }
        if (required > owners.length)
            changeRequirement(owners.length);
        require(required>=1)
        emit OwnerRemoval(owner);
    }
```

Listing 3.7: MultiSigWallet.sol (revised)

Status This issue has been fixed in the commit: 5c490c2.

3.5 Suggested transactionExists() in revokeConfirmation()/executeTransaction()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `MultiSigWallet`
- Category: Status Codes [11]
- CWE subcategory: CWE-391 [3]

Description

As mentioned in Section 3.5, ccTokens makes use of multi-sig wallets to mitigate possible risk from single centralized party. The `MultiSigWallet` implementation provides a solid codebase to meet the requirement with standard APIs for `submitTransaction()`, `addTransaction()`, `confirmTransaction()`, `revokeConfirmation()`, and `executeTransaction()`.

In the following, we show the code snippet of two related routines, i.e., `revokeConfirmation()` and `executeTransaction()`. As their names indicate, they are used to either revoke or execute an existing transaction. We notice that both routines ensure `ownerExists()`, `confirmed()`, and `notExecuted()`. While the `confirmed()` state implies the presence of transaction, we feel it is still better to validate the presence of transaction (via `transactionExists(transactionId)`) that is being revoked or executed.

```
217     /// @dev Allows an owner to revoke a confirmation for a transaction.
218     /// @param transactionId Transaction ID.
219     function revokeConfirmation(uint transactionId)
220     public
221     ownerExists(msg.sender)
222     confirmed(transactionId, msg.sender)
223     notExecuted(transactionId)
224     {
225         confirmations[transactionId][msg.sender] = false;
226         emit Revocation(msg.sender, transactionId);
227     }

229     /// @dev Allows anyone to execute a confirmed transaction.
230     /// @param transactionId Transaction ID.
231     function executeTransaction(uint transactionId)
232     public
233     ownerExists(msg.sender)
234     confirmed(transactionId, msg.sender)
235     notExecuted(transactionId)
236     {
237         if (isConfirmed(transactionId)) {
238             Transaction storage txn = transactions[transactionId];
239             txn.executed = true;
240             if (external_call(txn.destination, txn.value, txn.data.length, txn.data))
241                 emit Execution(transactionId);
242             else {
243                 emit ExecutionFailure(transactionId);
244                 txn.executed = false;
245             }
246         }
247     }
```

Listing 3.8: MultiSigWallet.sol

In the meantime, since the transaction execution requires making an external call (line 240), which requires possible `ether` as the payment of `txn.value`. Therefore, it is also suggested to mark related functions as `payable`. The related functions include `executeTransaction()` and `confirmTransaction()`.

Recommendation Apply the above suggestion to validate the transaction presence in `revokeConfirmation()` and `executeTransaction()`. An example revision is shown below.

```
    /// @dev Allows an owner to revoke a confirmation for a transaction.
    /// @param transactionId Transaction ID.
    function revokeConfirmation(uint transactionId)
    public
```

```
    ownerExists(msg.sender)
    transactionExists(transactionId)
    confirmed(transactionId, msg.sender)
    notExecuted(transactionId)
    {
        confirmations[transactionId][msg.sender] = false;
        emit Revocation(msg.sender, transactionId);
    }

    /// @dev Allows anyone to execute a confirmed transaction.
    /// @param transactionId Transaction ID.
    function executeTransaction(uint transactionId)
    public
    payable
    ownerExists(msg.sender)
    transactionExists(transactionId)
    confirmed(transactionId, msg.sender)
    notExecuted(transactionId)
    {
        if (isConfirmed(transactionId)) {
            Transaction storage txn = transactions[transactionId];
            txn.executed = true;
            if (external_call(txn.destination, txn.value, txn.data.length, txn.data))
                emit Execution(transactionId);
            else {
                emit ExecutionFailure(transactionId);
                txn.executed = false;
            }
        }
    }
```

Listing 3.9: MultiSigWallet.sol (revised)

Status This issue has been fixed in the commit: 5c490c2.

3.6 Trust Issue of Admin Keys Behind Custodian

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: ccTokenController
- Category: Security Features [7]
- CWE subcategory: CWE-287 [2]

Description

In ccTokens, there is a protocol-wide custodian in MemberMgr. This custodian plays a critical role in confirming/rejecting the minting requests from a registered merchant and actually minting ccTokens in Ethereum.

If we take a close look at `confirmMintRequest()`, this specific routine takes two arguments: `requestHash` and `amount`. The first argument is the `requestHash` unambiguously generated from an earlier merchant's mint request. However, the second argument is given by the custodian. If there is a possible collusion between a merchant and the custodian, the (subverted) minted operations could be detrimental to the entire `ccTokens` ecosystem.

```
    function confirmMintRequest(bytes32 requestHash, uint amount) external onlyCustodian
         returns (bool) {
        uint blockNo = block.number;
        Request memory request = getPendingMintRequest(requestHash);
        require(blockNo > request.requestBlockNo, "confirmMintRequest failed");

        require(blockNo - 20 >= request.requestBlockNo, "confirmMintRequest failed, wait
            for 20 blocks");
        uint seq = request.seq;
        require(controller.mint(request.requester, amount), "mint failed");
        mintRequests[seq].status = RequestStatus.APPROVED;
        mintRequests[seq].amount = amount;
        mintRequests[seq].confirmedBlockNo = blockNo;

        emit MintConfirmed(
            request.seq,
            request.requester,
            amount,
            request.btcAddress,
            request.btcTxId,
            blockNo,
            calcRequestHash(request)
        );
        return true;
    }
```

Listing 3.10: ccTokenController.sol

Instead of having a single EOA account as the custodian, an alternative is to make use of multi-sig wallets. To further eliminate the administration key concern, it may be required to transfer the role to a community-governed DAO. In the meantime, a timelock-based mechanism might also be applicable for mitigation.

Recommendation Promptly transfer the `custodian` privilege to an appropriate governance contract.

Status This issue has been fixed in the commit: 5c490c2. This commit ensures that the `custodian` role can only confirm/reject, not specify, the amount.

4 | Conclusion

In this audit, we have analyzed the `ccTokens` design and implementation. The system presents a unique offering in enabling seamless integration of each crypto asset (in other blockchains) into the Ethereum ecosystem. The current code base is well organized and 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.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [14, 15, 16, 17, 20].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [22] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

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

[3] MITRE. CWE-391: Unchecked Error Condition. https://cwe.mitre.org/data/definitions/391.html.

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

[5] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[6] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

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

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

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

[10] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

[14] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[15] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[16] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[17] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[19] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[20] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[21] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.

[22] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.