SMART CONTRACT AUDIT REPORT

for

SWITCHEO

Prepared By: Shuxiao Wang

Hangzhou, China

December 31, 2020

Document Properties

Client	Switcheo
Title	Smart Contract Audit Report
Target	Ethereum Deposit For Switcheo TradeHub
Version	1.0
Author	Xuxian Jiang
Auditors	Xudong Shao, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	December 31, 2020	Xuxian Jiang	Release Candidate
0.3	December 25, 2020	Xuxian Jiang	Additional Findings #2
0.2	December 23, 2020	Xuxian Jiang	Additional Findings #1
0.1	December 21, 2020	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related `Ethereum` deposit contracts for `Switcheo TradeHub`, 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 Switcheo TradeHub

`Switcheo TradeHub` is a purpose-built sidechain for trading. It is run by a decentralized network of public nodes that each hosts a copy of `Switcheo`'s order matching engine, validating every trade submission. After validation and execution of trades on `Switcheo TradeHub`, transactions will be broadcast in batches to be settled on Layer-1 blockchains such as `Bitcoin`, `Ethereum`, and `NEO`. The order matching engine is based on homemade, proven off-chain order matching engine, which has thus far been running on centralized servers hosted by `Switcheo`. The audited code is `Ethereum` deposit contracts for `Switcheo TradeHub`. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Ethereum Deposit For Switcheo TradeHub

Item	Description
Issuer	Switcheo
Website	https://www.switcheo.network
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 31, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audited contracts assumes a trusted cross-chain manager contract and the manager contract itself is not part of this audit.

- https://github.com/Switcheo/switcheo-tradehub-eth.git (6521a12)

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

- 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 accordingly classified into four categories, 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) [7], 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), 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 design and implementation of the `Ethereum` deposit contracts for `Switcheo TradeHub`. 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	1	■
Informational	2	■ ■
Total	4	

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, 1 low-severity vulnerability, and 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Improved Validation of _validateSignature()	Business Logic	
PVE-002	Low	Accommodation of approve() Idiosyncrasies	Business Logic	
PVE-003	Medium	Assumed Trust On Cross-Chain Manager	Security Features	
PVE-004	Informational	Improved Event Generation With Indexed Assets	Time and State	

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 Improved Validation of _validateSignature()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LockProxy`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

The deposit contracts for `Switcheo TradeHub` supports two forms of deposits: The first form is initiated from the depositing users and the funds are directly transferred by the users while the second one allows for transfers from the so-called `Wallet` contracts. For the second form, in order to validate that the users indeed authorize the transfer, the protocol provides the following `_validateLockFromWallet()` routine.

```
    /// @dev validate the signature for lockFromWallet
    function _validateLockFromWallet(
        address _walletOwner,
        address _assetHash,
        bytes memory _targetProxyHash,
        bytes memory _toAssetHash,
        bytes memory _feeAddress,
        uint256[] memory _values,
        uint8 _v,
        bytes32[] memory _rs
    )
        private
    {
        bytes32 message = keccak256(abi.encodePacked(
            "sendTokens",
            _assetHash,
            _targetProxyHash,
            _toAssetHash,
            _feeAddress,
            _values[0],
```

```
            _values[1],
            _values[2]
        ));

        require(seenMessages[message] == false, "Message already seen");
        seenMessages[message] = true;
        _validateSignature(message, _walletOwner, _v, _rs[0], _rs[1]);
    }
```

Listing 3.1: LockProxy::_validateLockFromWallet()

This routine essentially computes the message that has been signed by the user and calls a helper routine, i.e, `_validateSignature()`, for signature verification. Note that the adopted signature type is `EthSign` that basically has a prefixed message, `"\x19Ethereum Signed Message:\n32"`.

```
624     /// @dev validates a signature against the specified user address
625     function _validateSignature(
626         bytes32 _message,
627         address _user,
628         uint8 _v,
629         bytes32 _r,
630         bytes32 _s
631     )
632         private
633         pure
634     {
635         bytes32 prefixedMessage = keccak256(abi.encodePacked(
636             "\x19Ethereum Signed Message:\n32",
637             _message
638         ));
639 
640         require(
641             _user == ecrecover(prefixedMessage, _v, _r, _s),
642             "Invalid signature"
643         );
644     }
```

Listing 3.2: LockProxy::_validateSignature()

The verification is performed with `ecrecover()`, which is one of those pre-compiled contracts. The idea here is to compute the public key corresponding to the private key that was used to create an `ECDSA` signature. It returns the recovered address associated with the public key or returns zero on error. With that, the above `_validateSignature()` can be improved by further enforcing the following requirement: `require(_user != address(0), "Invalid signature")`.

Recommendation Properly handle the situation when the underlying `ecrecover()` routine returns zero on error. It should be noted that the `_user` (line 641) is the owner of the wallet, which has guaranteed to be non-$address(0)$ in all possible execution paths.

Status

3.2 Accommodation of approve() Idiosyncrasies

- ID: PVE-002
- Severity: low
- Likelihood: low
- Impact: medium
- Target: LockProxy
- Category: Business Logic [6]
- CWE subcategory: N/A

Description

The LockProxy contract is flexible in specifying so-called extensions. A new extension can be added via the addExtension() method through the governance on Switcheo TradeHub. Also, an existing one can be removed via the removeExtension() method also through the governance on Switcheo TradeHub . Note the authorized extension can transfer assets out of the LockProxy contract, including ETH or other approved tokens.

To elaborate, we show below the extensionTransfer() routine. Through this routine, an authorized extension can directly move funds out of the contract. We note that the transfer of approved tokens takes the two steps. The first step in essence calls approve() (lines 405−412) to specify the allowance of the spender, i.e., _receivingAddress. And the second step will require _receivingAddress to call transferFrom() to actually transfer the fund.

```
302     function extensionTransfer(
303         address _receivingAddress,
304         address _assetHash,
305         uint256 _amount
306     )
307         external
308         returns (bool)
309     {
310         require(
311             extensions[msg.sender] == true,
312             "Invalid extension"
313         );

315         if (_assetHash == ETH_ASSET_HASH) {
316             // we use 'call' here since the _receivingAddress could be a contract
317             // see https://diligence.consensys.net/blog/2019/09/stop-using-soliditys-
                    transfer-now/
318             // for more info
319             (bool success, ) = _receivingAddress.call{value: _amount}("");
320             require(success, "Transfer failed");
321             return true;
322         }

324         ERC20 token = ERC20(_assetHash);
325         _callOptionalReturn(
```

```
            token,
            abi.encodeWithSelector(
                token.approve.selector,
                _receivingAddress,
                _amount
            )
        );

        return true;
    }
```

Listing 3.3: LockProxy::extensionTransfer()

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 this following, we examine the `approve()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.4: USDT Token **Contract**

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. As a result, in order to accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Note that the accommodation of the `approve()` idiosyncrasy is necessary to ensure a smooth `extensionTransfer()`. Otherwise, the extension transfer attempt with inconsistent token contracts may always be reverted.

Recommendation Accommodate the above-mentioned idiosyncrasy of `approve()`.

Status

3.3 Assumed Trust On Cross-Chain Manager

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `LockProxy`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the developed `Ethereum` deposit contracts for `Switcheo TradeHub`, there is a privileged contract, i.e., `CCM`, that plays a critical role in configuring and regulating the system-wide operations (e.g., `extension` adjustment, `asset` approvals, and funds-unlocking). Note the unlocking of funds directly affects the user deposits.

In the following, we show the contract's `unlock()` implementation. By design, this routine simply follows the instructions to release funds to the intended recipient, i.e., `toAddress` (line 369). Note the associated `onlyManagerContract` modifier restricts this call can only be invoked by the authorized `CCM`.

```
344     /// @dev Performs a withdrawal that was initiated on Switcheo TradeHub
345     /// @param _argsBz the serialized TransferTxArgs
346     /// @param _fromContractAddr the associated contract address on Switcheo TradeHub
347     /// @param _fromChainId the originating chainId
348     /// @return true if success
349     function unlock(
350         bytes calldata _argsBz,
351         bytes calldata _fromContractAddr,
352         uint64 _fromChainId
353     )
354         external
355         onlyManagerContract
356         nonReentrant
357         returns (bool)
358     {
359         require(_fromChainId == counterpartChainId, "Invalid chain ID");
360 
361         TransferTxArgs memory args = _deserializeTransferTxArgs(_argsBz);
362         require(args.fromAssetHash.length > 0, "Invalid fromAssetHash");
363         require(args.toAssetHash.length == 20, "Invalid toAssetHash");
364 
```

```
        address toAssetHash = Utils.bytesToAddress(args.toAssetHash);
        address toAddress = Utils.bytesToAddress(args.toAddress);

        _validateAssetRegistration(toAssetHash, _fromContractAddr, args.fromAssetHash);
        _transferOut(toAddress, toAssetHash, args.amount);

        emit UnlockEvent(toAssetHash, toAddress, args.amount, _argsBz);
        return true;
    }
```

Listing 3.5: LockProxy::unlock()

We emphasize that the current privilege assignment to `CCM` is appropriate and necessary. However, it is worrisome if `CCM` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that the governance will be managed by a multisig account. 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.

We point out that a compromised `CCM` account would either directly transfer funds out or allow the attacker to add a malicious `extension` to steal all funds in the contract. Either way would directly undermine the integrity of the entire protocol.

Recommendation Promptly transfer the `CCM` privilege to the intended `DAO`-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status

3.4 Improved Event Generation With Indexed Assets

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LockProxy`
- Category: Time and State [5]
- CWE subcategory: CWE-362 [2]

Description

Meaningful `events` are an important part in smart contract design as they can not only greatly expose the runtime dynamics of smart contracts, but also allow for better understanding about their behavior and facilitate off-chain analytics. The `events` are typically emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed.

We examine the use of `event` and pay attention to key operations in the protocol. In the following, we list a few representative events that have been defined in the deposit contracts.

```
    event LockEvent(
        address fromAssetHash,
        address fromAddress,
        uint64 toChainId,
        bytes toAssetHash,
        bytes toAddress,
        bytes txArgs
    );

    event UnlockEvent(
        address toAssetHash,
        address toAddress,
        uint256 amount,
        bytes txArgs
    );
```

Listing 3.6: Main Events Defined in LockProxy

It comes to our attention that the above list of events makes no use of `indexed` in the emitted address information. Note that each emitted event is represented as a topic that usually consists of the signature (from a `keccak256` hash) of the event name and the types (`uint256`, `string`, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, which means it will be attached as data (instead of a separate topic). Considering that the asset/address information is typically queried, it is better treated as a topic, hence the need of being `indexed`.

Recommendation Revise the above events by properly indexing the emitted asset/address information.

Status

4 Conclusion

In this audit, we have analyzed the documentation and implementation of the `Ethereum` deposit contracts for `Switcheo TradeHub`, which is a purpose-built sidechain for trading. The audited system presents a reliable cross-chain, integrated component for `Switcheo TradeHub`. We are impressed by the overall design and solid implementation. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

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

[5] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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