

# Security Audit

## of SWITCHEO's Smart Contracts

November 15, 2018












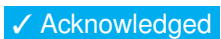


Produced for



by



# Table Of Content

Foreword . . . . .	1
Executive Summary . . . . .	1
Audit Overview . . . . .	2
1. Scope of the Audit . . . . .	2
2. Depth of Audit . . . . .	2
3. Terminology . . . . .	2
Limitations . . . . .	4
System Overview . . . . .	5
1. System actors . . . . .	5
2. System operation . . . . .	5
3. System states . . . . .	5
Best Practices in SWITCHEO's project . . . . .	6
1. Hard Requirements . . . . .	6
2. Soft Requirements . . . . .	6
Security Issues . . . . .	7
1. User can deposit and pay fees in fake ERC20 tokens   . . . . .	7
2. Depositing and trading allowed in <code>Inactive</code> state   . . . . .	7
3. Outdated compiler version   . . . . .	7
4. <code>onlyOwner</code> modifier checks wrong address   . . . . .	7
5. Non-whitelisted user can approve spenders   . . . . .	8
6. Use of <code>extcodesize</code>   . . . . .	8
Trust Issues . . . . .	9
1. Non-transferable ownership   . . . . .	9

2.	coordinator allowed to transfer ERC20 from user's wallet	L	✓ Acknowledged	9
3.	coordinator allowed to cancel announcedCancellations	L	✓ Addressed	9
Design Issues . . . . .				10
1.	Missing input validations	L	✓ Fixed	10
2.	Unbounded iterations in fillOffers function	L	✓ Acknowledged	10
3.	coordinator allowed to transfer ERC20 from user's wallet	L	✓ Acknowledged	10
4.	Ordering of variables in struct can be optimized	L	✓ Fixed	10
Recommendations / Suggestions . . . . .				12
Disclaimer . . . . .				13

# Foreword

We first and foremost thank SWITCHEO for giving us the opportunity to audit their smart contracts. This document outlines our methodology, limitations, and results.

– ChainSecurity

## Executive Summary

The SWITCHEO smart contracts have been analyzed under different aspects, with a variety of tools for automated security analysis of Ethereum smart contracts and expert manual review.

Overall, we found that SWITCHEO employs good coding practices and has a clean code base. Nonetheless, CHAINSECURITY was able to uncover several security, design and trust issues that were successfully mitigated or addressed by SWITCHEO before deployment.

# Audit Overview

## Scope of the Audit

The scope of the audit is limited to the following source code files. All of these source code files were received on October 3, 2018 and an updated version on November 14, 2018:

File	SHA-256 checksum
Broker.sol	4696b37a35d015ab58234607ec4ecc3958347ed47261a49202738f1d73ecc8e0

## Depth of Audit

The scope of the security audit conducted by CHAINSECURITY was restricted to:

- Scan the contracts listed above for generic security issues using automated systems and manually inspect the results.
- Manual audit of the contracts listed above for security issues.

## Terminology





For the purpose of this audit, we adopt the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology<sup>1</sup>).

**Likelihood** represents the likelihood of a security vulnerability to be encountered or exploited in the wild.










**Impact** specifies the technical and business related consequences of an exploit.

**Severity** is derived based on the likelihood and the impact calculated previously.

We categorize the findings into 4 distinct categories, depending on their severities:

-  Low: can be considered as less important
-  Medium: should be fixed
-  High: we strongly suggest to fix it before release
-  Critical: needs to be fixed before release

These severities are derived from the likelihood and the impact using the following table, following a standard approach in risk assessment.

LIKELIHOOD	IMPACT		
	High	Medium	Low
High			
Medium			
Low			

<sup>1</sup>[https://www.owasp.org/index.php/OWASP\\_Risk\\_Rating\\_Methodology](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology)

During the audit concerns might arise or tools might flag certain security issues. If our careful inspection reveals no security impact, we label it as **✓ No Issue**. If during the course of the audit process, an issue has been addressed technically, we label it as **✓ Fixed**, while if it has been addressed otherwise by improving documentation or further specification, we label it as **✓ Addressed**. Finally, if an issue is meant to be fixed in the future without immediate changes to the code, we label it as **✓ Acknowledged**.

Findings that are labelled as either **✓ Fixed** or **✓ Addressed** are resolved and therefore pose no security threat. Their severity is still listed, but just to give the reader a quick overview what kind of issues were found during the audit.

# Limitations

Security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, auditing allows to discover vulnerabilities that were overlooked during development and areas where additional security measures are necessary.

In most cases, applications are either fully protected against a certain type of attack, or they lack protection against it completely. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. We therefore carry out a source code review trying to determine all locations that need to be fixed. Within the customer-determined timeframe, CHAINSECURITY has performed auditing in order to discover as many vulnerabilities as possible.

# System Overview

SWITCHEO is creating a Decentralised Exchange (DEX). In this exchange users can trade ETH and any ERC20 tokens. Users can deposit their ETH and/or ERC20 tokens to the `Broker` contract to trade on the exchange.

## System actors

A user refers to an entity that interacts with the `Broker` contract. The following types of roles participate in the SWITCHEO exchange:

- **Owner:** Owner of the `Broker` contract. Has the power to update the `coordinator` and `operator` addresses and controls the current state of the exchange.
- **Coordinator:** Sends trades to the exchange on behalf of the `Maker` or `Filler`.
- **Operator:** All the fees paid by the `Maker` or `Filler` while sending transactions are collected at the `operator` address.
- **Makers:** Users who create new offers on the DEX.
- **Fillers:** User who fill offers.

## System operation

To initiate transactions, `Maker/Filler` has to sign the trade data off-chain. This signed data is sent to SWITCHEO and the `Coordinator` initiates the transaction on the DEX on users' behalf.

Users are allowed to do following operations:

- Deposit ETH to the `Broker` contract.
- Deposit any ERC20 token.
- Withdraw ETH / ERC20 tokens.
- `Maker` can create new trade offers.
- `Filler` can fill one or multiple offers.
- `Maker` can cancel an offer he created.

## System states

The broker contract can be in two different states:

- **Active:** All deposits and trades are allowed for the users. `Coordinator` can call deposit and trade transactions on behalf of user.
- **Inactive:** Only the `Coordinator` is allowed to act while the exchange is in this state. The allowed operations are canceling an offer or forcefully withdrawing user's funds. In the later case funds are directly sent to the user's wallet.



# Best Practices in SWITCHEO's project

Projects of good quality follow best practices. In doing so, they make audits more meaningful, by allowing efforts to be focused on subtle and project-specific issues rather than the fulfillment of general guidelines.

Avoiding code duplication is a good example of a good engineering practice which increases the potential of any security audit.

We now list a few points that should be enforced in any good project that aims to be deployed on the Ethereum blockchain. The corresponding box is ticked when SWITCHEO's project fitted the criterion when the audit started.

## Hard Requirements

These requirements ensure that the SWITCHEO's project can be audited by CHAINSECURITY.

- ☒ The code is provided as a Git repository to allow the review of future code changes.
- ☒ Code duplication is minimal, or justified and documented.
- ☒ Libraries are properly referred to as package dependencies, including the specific version(s) that are compatible with SWITCHEO's project. No library file is mixed with SWITCHEO's own files.
- ☒ The code compiles with the latest Solidity compiler version. If SWITCHEO uses an older version, the reasons are documented.
- ☒ There are no compiler warnings, or warnings are documented.

## Soft Requirements

Although these requirements are not as important as the previous ones, they still help to make the audit more valuable to SWITCHEO.

- ☒ There are migration scripts.
- ☒ There are tests.
- ☐ The tests are related to the migration scripts and a clear separation is made between the two.
- ☒ The tests are easy to run for CHAINSECURITY, using the documentation provided by SWITCHEO.
- ☒ The test coverage is available or can be obtained easily.
- ☒ The output of the build process (including possible flattened files) is not committed to the Git repository.
- ☐ The project only contains audit-related files, or, if not possible, a meaningful separation is made between modules that have to be audited and modules that CHAINSECURITY should assume correct and out of scope.
- ☒ There is no dead code.
- ☒ The code is well documented.
- ☒ The high-level specification is thorough and allow a quick understanding of the project without looking at the code.
- ☒ Both the code documentation and the high-level specification are up to date with respect to the code version CHAINSECURITY audits.
- ☒ There are no getter functions for public variables, or the reason why these getters are in the code is given.
- ☒ Function are grouped together according either to the Solidity guidelines<sup>2</sup>, or to their functionality.

---

<sup>2</sup><https://solidity.readthedocs.io/en/latest/style-guide.html#order-of-functions>

# Security Issues

In the following, we discuss our investigation into security issues. Therefore, we highlight whenever we found specific issues but also mention what vulnerability classes do not appear, if relevant.

## User can deposit and pay fees in fake ERC20 tokens

A user can deposit any token in the `Broker` contract by passing it as a parameter to the `depositERC20` function. However, there is no mechanism in place to prevent depositing a fake ERC20 token, which has no value. This can later be abused by a malicious user who can pay fees to the coordinator in the form of the fake token.

SWITCHEO must ensure that the ERC20 token address being deposited is supported by DEX.

**Likelihood:** Medium

**Impact:** Medium

**Acknowledged:** SWITCHEO states that they are validating trade requests off-chain and `Coordinator` will process only those requests which have token addresses that SWITCHEO accepts as fees. Other trade requests with fake ERC20 token addresses will be discarded by their off-chain trade server.

## Depositing and trading allowed in Inactive state

The `Inactive` state allows emergency methods to be called by the `coordinator`. However, no other methods related to deposits and trading should be callable while in the `Inactive` state. However, currently depositing and trading is possible in both, the `Active` and `Inactive` state.

**Likelihood:** Medium

**Impact:** Medium

**Fixed:** SWITCHEO solved the problem by introducing modifiers `onlyActiveState` and `onlyInactiveState` and updating the corresponding functions.

## Outdated compiler version

Without a documented reason, the newest compiler version should be used homogeneously. In particular, given the vulnerabilities of version 0.4.24, SWITCHEO should enforce the new version 0.4.25 through the use of a corresponding `pragma`.

**Likelihood:** Medium

**Impact:** Medium

**Fixed:** SWITCHEO solved this by using the new compiler version 0.4.25 in `Broker`.

## onlyOwner modifier checks wrong address

The `Broker` contract implements a standard `onlyOwner` modifier. However, instead of checking whether the `msg.sender` is the owner of the contract, the modifier requires that the sender is the `coordinator` of the contract.

```
modifier onlyOwner() {
    require(
        msg.sender == coordinator,
        "Invalid sender"
    );
    _;
}
```

As a result, the owner gets prevented from executing owner specific operations, while the coordinator role gets higher privileges.

**Likelihood:** Medium

**Impact:** High

**Fixed:** SWITCHEO solved this problem by inheriting from `Claimable` and the `onlyOwner` function is removed from `Broker`.

#### Non-whitelisted user can approve spenders

A user that is not whitelisted himself, is still able to call `approveSpender` and pass an address of a whitelisted user that will be able to trade on his behalf, thereby circumventing KYC requirements on his account.

Such a malicious user, having obtained tokens without going through the KYC process (e.g. obtaining them on secondary markets or stealing them), will find a whitelisted partner, call `approveSpender` on his partner's address and is henceforth able to trade his questionably obtained tokens directly on his own account through the SWITCHEO exchange.

**Likelihood:** Medium

**Impact:** Medium

**Addressed:** SWITCHEO addresses the issue by only whitelisting spenders that are other audited smart contracts launched by SWITCHEO or its partners. These contracts should not allow users to bypass any KYC checks.

#### Use of `extcodesize`

Function `_validateIsContract` defined in the `Broker` contract, makes use of the `extcodesize` assembly instruction to check the size of an address that is passed as a parameter. The function is then used in several places throughout the code to verify that an address is a contract.

```
function _validateIsContract(address addr) private view {
    assembly {
        if iszero(extcodesize(addr)) { revert(0, 0) }
    }
}
```

Although these usages impose no security threat to SWITCHEO's code, CHAINSECURITY would like to raise awareness about a potential vulnerability of this coding pattern. If `extcodesize` is used to enforce that an address is an externally owned account as opposed to a contract account, the check can be easily circumvented. A contract can pretend to be an EOA by calling a function from its constructor, which would result in a `extcodesize(_addr) == 03`.

**Likelihood:** Low

**Impact:** Low

**Acknowledged:** SWITCHEO acknowledges the potential issues regarding the use of assembly instructions, but will not make changes since there is no impact on the contract's security.

<sup>3</sup><https://consensys.github.io/smart-contract-best-practices/recommendations/#avoid-using-extcodesize-to-check-for-externally->

# Trust Issues

The issues described in this section are not security issues but describe functionality which is not fixed inside the smart contract and hence requires additional trust into SWITCHEO, including in SWITCHEO's ability to deal with such powers appropriately.

## Non-transferable ownership

The owner of the `Broker` contract is assigned in the constructor and cannot be changed after deployment. owner privileges include changing addresses like `coordinator` and `operator`. In case the private key of the owner is stolen, a malicious user could assign any address to those two roles.

CHAINSECURITY recommends using `Ownable` or `Claimable` from OpenZeppelin release v1.12.0<sup>4</sup>. This would allow to control the `Broker` contract using a MultiSig.

**Fixed:** SWITCHEO solved the problem by inheriting from `Claimable` by OpenZeppelin.

## `coordinator` allowed to transfer ERC20 from user's wallet

The `coordinator` is able to execute a withdrawal from a user's token wallet immediately after the user has approved a transfer to the `Broker` contract.

This makes the function call inconsistent with the rest of the `Broker` interface. All other transactions need to be explicitly signed off-chain by the user, for the `coordinator` to be allowed to execute them. Therefore, `depositERC20` is the only function which can be called without an explicit permission from the user.

**Acknowledged:** SWITCHEO acknowledges this.

## `coordinator` allowed to cancel `announcedCancellations`

A `coordinator` can make a call to the `fastCancel` function and cancel the offer of any offer maker, leaving him without the possibility to not cancel by avoiding the call to `slowCancel()`. The only prerequisite limiting the `coordinator` is that the `maker` must have previously called `announceCancellation()` for their `offerHash`.

**Addressed:** SWITCHEO clarifies that the `announcedCancel` operation is intended to be an announcement of intent to cancel, so there is no trust violation in helping the user cancel the offer ahead of time. The `slow*` operations are intended to allow users to cancel and withdraw funds in the event the `coordinator` is byzantine and stops broadcasting transactions, and should not be used in normal cases.

<sup>4</sup> <https://github.com/OpenZeppelin/openzeppelin-solidity/releases/tag/v1.12.0>

# Design Issues

The points listed here are general recommendations about the design and style of SWITCHEO's project. They highlight possible ways for SWITCHEO to further improve the code.

## Missing input validations

Coordinator and operator addresses need to be checked that they are not accidentally set to the zero address by the owner.

```
function setCoordinator(address _coordinator) external onlyOwner {
    require(_coordinator != address(0));
    coordinator = _coordinator;
}
```

Although the owner can change the address as soon as he spots the mistake, some tokens can still get burnt in the meantime by paying fees to the zero address.

**Fixed:** SWITCHEO solved the problem by introducing a method `_validateAddress` and ensuring input validations of address.

## Unbounded iterations in `fillOffers` function

`fillOffers()` loops through the array of offers to be filled. However, there are no bounds on the size of the array, which means there is a possibility that the transaction exceeds the block gas limit if the user tries to fill a large number of offers.

In those cases the transaction will revert and it will be up to the user to create and sign two `fillOffers()` calls. Only then the coordinator can verify the new hashes and execute the transactions.

SWITCHEO should ensure that the number of hashes passed to `fillOffers()` is limited so the transaction cost remains under the block gas limit. Alternatively, the user should at least be aware that the execution can fail under the mentioned conditions.

**Acknowledged:** SWITCHEO acknowledged that they will design their off-chain APIs around this so that the number of offers will be limited not to exceed block gas limit.

## coordinator allowed to transfer ERC20 from user's wallet

The coordinator is able to execute a withdrawal from a user's token wallet immediately after the user has approved a transfer to the Broker contract.

This makes the function call inconsistent with the rest of the Broker interface. All other transactions need to be explicitly signed off-chain by the user, for the coordinator to be allowed to execute them. Therefore, `depositERC20` is the only function which can be called without an explicit permission from the user.

**Acknowledged:** SWITCHEO acknowledges this.

## Ordering of variables in `struct` can be optimized

The current ordering of variables inside the `Offer` `struct` is not optimal. To allow EVM optimizations, members in a `struct` have to be packed in a way that shorter types are grouped together.

```
struct Offer {
    address maker;
    address offerAsset;
    address wantAsset;
    uint64 nonce;
```

```
uint256 offerAmount;  
uint256 wantAmount;  
uint256 availableAmount; // the remaining offer amount  
}
```

The storage of Ethereum is organized in slots of 256 bits. That is why an `address` variable (160 bits) can be packed together with a `uint64` variable (64 bits) into a single slot. In the current implementation, the `nonce` variable is declared after a `uint256` and uses a whole slot by itself, despite needing only 64 bits.

**Fixed:** SWTCHEO has solved the problem in the code and adopted recommended variable ordering in struct.

# Recommendations / Suggestions

- ✓ Both `depositEther` and `depositERC20` functions allow deposits of 0 Ether or tokens. If SWITCHEO does not want such deposits to get executed, then the following checks could be implemented:

```
require(msg.value > 0);
```

and

```
require(_amount > 0);
```

- ✓ An extra `require` statement could be used to check the length of the `_offerHashes` array, which is passed as an argument to the `fillOffers` function:

```
require(_offerHashes.length > 0)
```

This will ensure that at least one offer hash is present and protect the user from paying fees when there are no offers to be filled.

- ✓ The following snippet of code is used in three places in `Broker.sol`.

```
require(
    usedHashes[msgHash] != true,
    "msgHash_already_used"
);

usedHashes[msgHash] = true;
```

CHAINSECURITY recommends refactoring this to a single `private` function that gets reused instead of duplicating code. For example an appropriate name and signature for such a function could be:

```
verifyAndAddHash(bytes32 _msgHash) {...}
```

- ✓ `offerHash` can also be indexed in the following events which would allow searching for events with `offerHash` as well.

```
event Make(address indexed maker, bytes32 offerHash);
event Fill(address indexed filler, bytes32 offerHash, uint256 amountFilled
    , uint256 amountTaken, address indexed maker);
event Cancel(address indexed maker, bytes32 offerHash);
```

- ✓ `makeOffer` function can use the signed `msgHash` instead of re-creating `offerHash` again.
- ✓ While SWITCHEO has descriptive variable and functions names, code readability and documentation can be significantly improved by commenting the code which is currently only done in few spots.
- ✓ CHAINSECURITY recommends that SWITCHEO should use double quotes for string literals. This maintains consistency with the other string literals present in code.
- ✓ CHAINSECURITY applauds the efforts SWITCHEO has made to exhaustively comment the code. In the process, some minor typing mistakes were introduced that can be easily fixed, such as `faciliates`, `rescient`, `priviledges` and `autoamtically`.

**Post-audit comment:** SWITCHEO has fixed all of the above issues. Therefore, SWITCHEO has to perform no more code changes with regards to these recommendations.

# Disclaimer

UPON REQUEST BY SWITCHEO, CHAINSECURITY LTD. AGREES MAKING THIS AUDIT REPORT PUBLIC. THE CONTENT OF THIS AUDIT REPORT IS PROVIDED "AS IS", WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND, AND CHAINSECURITY LTD. DISCLAIMS ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT. COPYRIGHT OF THIS REPORT REMAINS WITH CHAINSECURITY LTD..