CONFIDENTIAL

Security Audit

of SetProtocol's Smart Contracts

February 18, 2019

Produced for

by

Table Of Content

Foreword . . . 1

Executive Summary . . . 1

Audit Overview . . . 2

1. Methodology and Scope of the Audit . . . 2

2. Depth of Audit . . . 2

3. Scope . . . 2

4. Terminology . . . 3

Limitations . . . 4

Best Practices in SETPROTOCOL's project . . . 5

1. Hard Requirements . . . 5

2. Soft Requirements . . . 5

Verified Properties . . . 6

1. Properties related to the SETPROTOCOL contracts . . . 6

1.1 Withdrawing always reduces balance ✓ Verified . . . 6

1.2 All access to Vault functions must be authorized ✓ Verified . . . 6

Security Issues . . . 7

1. There haven't been any security issues uncovered in the code under audit. . . . 7

Trust Issues . . . 7

1. There haven't been any trust issues uncovered in the code under audit. . . . 7

Design Issues . . . 7

1. Attempt to block ether flows L ✓ Acknowledged . . . 7

2. Unbounded iterations M ✓ Acknowledged . . . 7

3. Redundant inheritance in `Core` L ✓ Fixed . . . 8

4. Suboptimal struct layout L ✓ Fixed . . . 8

5. Silent overrides of mapping entries M ✓ Fixed 9

6. Unused function parameter L ✓ Fixed 9

7. Function state mutability L ✓ Fixed 9

8. Unnecessary conversion L ✓ Fixed 10

Recommendations / Suggestions **11**

Foreword

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

– ChainSecurity

Executive Summary

The SETPROTOCOL 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 SETPROTOCOL employs very good coding practices and has clean and exceptionally well-commented code, including clear functional specifications and exhaustive documentation for all parts of its system.

CHAINSECURITY performed a narrowly scoped audit within a short time-frame, which given the complexity and size of the system cannot be considered complete.

CHAINSECURITY worked towards formally verifying a set of properties and included the ones which could be verified successfully. Due to the complexity of the system, more advanced properties couldn't be verified in the available time.

Several low- and medium-severity design issues were uncovered. SETPROTOCOL addressed all issues attentively, CHAINSECURITY has no remaining design or security concerns.

Audit Overview

Methodology and Scope of the Audit

CHAINSECURITY's methodology in performing the security audit consists of four chronologically executed phases:

1. Understanding the existing documentation, purpose, and specifications of the smart contracts.
2. Executing automated tools to scan for generic security vulnerabilities.
3. Manual analysis covering both functional (best effort based on the provided documentation) and security aspects of the smart contracts by one of our CHAINSECURITY experts.
4. Writing the report with the individual vulnerability findings and potential exploits.

The scope of the audit is limited to the following source code files. All of these source code files were received on January 16, 2019[1] and updated versions on February 19, 2019:

File	SHA-256 checksum
./core/Vault.sol	ecd7be220ff0c817fd162de6533273af520e4cbbf3836a4d911ae59b886fbd8d
./core/interfaces/IVault.sol	9635fecca656e4ef0e0dfdc567f9bdf8ed835f4581ba86262b148cae689c69a2
./core/lib/CoreState.sol	1d4fe3cbdd05d53ec7b797d9e98b6ce07558ee8530a2c954442ffc49b1a88657
./core/extensions/CoreInternal.sol	2ec16603b1aac709ae2879c2676cd1b8b78bf8751d88a8d4d807a2690f25cf26
./core/extensions/CoreFactory.sol	9b5b3904a164b2310b95532bc303c7807ee72a4730149356ee9f6b2f69ab946d
./core/interfaces/ICoreAccounting.sol	047e3d659ea7dcb75dea7499ca5ccb847bdc6fdba88267f39330563ca7675642
./core/extensions/CoreAccounting.sol	16605332c6f62aedb1b839405da28ab354490165ee1c52e894cf6cd4ceff425b
./core/modules/lib/ModuleCoreState.sol	e9ef40687933896c7b0612bce6288396b475d77a6081979252b1ddaf35479295
./core/interfaces/IExchangeIssueModule.sol	a5ef347f3bc82b85058625e1b6d9cd014896c5fca3d319eaf0cef4187e334dde
./core/modules/ExchangeIssueModule.sol	020e939ec500445a82c7442a07622c4c28c39275561b5aef843ea98e2252e5e6
./core/modules/RebalancingTokenIssuanceModule.sol	75c219e883db519f13d16989d1b04f929986d6098fb5c16dd88ccff8e76fbbcc
./supplementary/PayableExchangeIssue.sol	6159cd32d4cc83af930e3f33ec1b4a628cb9de97a3ebd1f70f09de9e2f43935e
./supplementary/rebalancing-manager/BTCETHRebalancingManager.sol	eb3b2579a5d896a7b46a2b92cd61435e6e77fcf29a9fc719ffc4251202da3afa

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.

Scope

SETPROTOCOL requested a precisely scoped audit, meant to assess the technical foundation of SETPROTOCOL's project in its current state within a narrow range.

Issues that have been encountered while verifying these and other interacting contracts have been listed, even when they were not explicitly included in the scope. However, this scope should not be considered exhaustive with respect to the security of SETPROTOCOL's smart contracts. CHAINSECURITY strove to verify the key functionality of the SETPROTOCOL system's core, to provide a report which contents could serve as potential guidelines in the future and highlight shortcomings in the current state.

[1] https://github.com/SetProtocol/set-protocol-contracts/tree/c12cc0ce91f3aebbe489bf02989148f3d12ab1fe

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[2]).

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:

- L Low: can be considered as less important
- M Medium: should be fixed
- H High: we strongly suggest to fix it before release
- C 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.

	IMPACT		
LIKELIHOOD	**High**	**Medium**	**Low**
High	C	H	M
Medium	H	M	L
Low	M	L	L

During the audit concerns might arise or tools might flag certain security issues. After careful inspection of the potential security impact, we assign the following labels:

- ✓ No Issue: no security impact
- ✓ Fixed: during the course of the audit process, the issue has been addressed technically
- ✓ Addressed: issue addressed otherwise by improving documentation or further specification
- ✓ Acknowledged: issue is meant to be fixed in the future without immediate changes to the code

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.

[2] https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

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.

Best Practices in SETPROTOCOL'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 SETPROTOCOL's project fitted the criterion when the audit started.

Hard Requirements

These requirements ensure that the SETPROTOCOL'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 SETPROTOCOL's project. No library file is mixed with SETPROTOCOL's own files.
- ☐ The code compiles with the latest Solidity compiler version. If SETPROTOCOL 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 SETPROTOCOL.

- ☑ 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 SETPROTOCOL.
- ☑ 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[3], or to their functionality.

[3] https://solidity.readthedocs.io/en/latest/style-guide.html#order-of-functions

Verified Properties

CHAINSECURITY formalized and inspected relevant correctness properties, such as invariants that must hold for the lifetime of the contract. Below, the verified and violated properties are listed:

Properties related to the SETPROTOCOL contracts

1.1 Withdrawing always reduces balance ✓ Verified

The vault always ends up with reduced balance for a particular ERC20 token after a call to `withdrawTo()`.

```
always (
    ((FUNCTION == Vault.withdrawTo(address,address,uint256)) && (Vault.
       withdrawTo(address,address,uint256)[0] == Vault.test_token) && (Vault.
       authorized[msg.sender] == true)) ==>
    (prev(ERC20._balances[Vault]) <= ERC20._balances[Vault])
);
frame(Vault.test_token == ERC20);
```

1.2 All access to Vault functions must be authorized ✓ Verified

Every function in Vault is only callable by an authorized core contract.

```
always((true==prev(Vault.authorized[msg.sender]))||(msg.sender==prev(Vault.
    _owner)));
```

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.

There haven't been any security issues uncovered in the code under audit.

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 SetProtocol, including in SetProtocol's ability to deal with such powers appropriately.

There haven't been any trust issues uncovered in the code under audit.

Design Issues

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

Attempt to block ether flows L ✓ Acknowledged

The `PayableExchangeIssue` contract tries to block incoming ETH through a `require` clause in the fallback function. However, it should be noted that ETH can always be forced into a contract through a `selfdestruct` or mining to that address[4]. While it seems that further no assumptions are based on this contract's balance, should this not hold true for any part of the off-chain system, ChainSecurity recommends to review these assumptions and consider introducing ETH handling capabilities separate from the `private` `returnExcessFunds` function.

Acknowledged: SetProtocol determined that forcing ETH into the contract is unlikely to have adverse effects to the system.

Unbounded iterations M ✓ Acknowledged

Contract `Vault` defines several batch functions that allow an authorized `Core` address to execute `withdraw`, `increment`, `decrement` and `transfer` on an address' balance for several tokens at once. This operation is executed in a `for` loop whose iterations depend on the length of the `_tokens` array.

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. In those cases the transaction will revert and it will be up to the user of the contract to create separate smaller function calls.

SetProtocol should ensure that the number of entries passed to any function iterating over an indefinite length 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.

Other occurrences of this malpractice can be found in:

- `ExchangeIssueModule`: `executeExchangeOrders()` loops over `orderData`, although this is safe under the assumption that there will always be some strict bound on the length of data in exchange orders that will not let the function hit the block gas limit.

[4] https://solidity.readthedocs.io/en/latest/security-considerations.html?highlight=selfdestruct

- `ExchangeIssueModule`: `calculateRequiredTokenBalances()` loops over the components of a Set, which makes sense to not be strictly limited, but a bound might be helpful.
- `StandardSettleRebalanceLibrary`: `calculateMaxIssueAmount()`. Here the remarks mentioned in the previous point apply as well.
- `RebalancingHelperLibrary`: `createTokenFlowArrays()` cycles through each token in `combinedTokenArray`. Should these be too many, the arrays of token in- and outflows will not be created.
- `RebalanceAuctionModule`: `withdrawFromFailedRebalance` iterates over a Set token's components. An iteration exceeding the gas limit here will block token withdrawals in case of a failed auction.

Given the importance of the functionality implemented in some of the functions from the preceding list, CHAINSECURITY recommends SETPROTOCOL to perform gas testing with tools such as eth-gas-reporter[5] to at least know what quantities lead to failures if the iterations cannot be bounded otherwise. Such knowledge will be useful to SETPROTOCOL and other actors interacting with its system.

Fixed: SETPROTOCOL acknowledges that they are aware of these unbounded iterations and states that they decided to let the gas limit dictate the limitations for these loops. Futhermore SETPROTOCOL adds that extensive profiling of gas costs of various transactions have been performed.

Redundant inheritance in `Core` L ✓ Fixed

Contract `Core` explicitly inherits `CoreState`. However, other parent contracts of `Core` such as `CoreFactory` and `CoreIssuance` already inherit `CoreState`, therefore making the explicit inheritance redundant and introducing a more convoluted inheritance scheme. This makes it harder to reason about the linearization process and potentially keep track of the contract's most derived function implementations.

Fixed: SETPROTOCOL removed the redundant inheritance in `Core.sol`.

Suboptimal struct layout L ✓ Fixed

Fields in struct `State` defined in contract `CoreState` can be reordered to optimize gas consumption.

```
struct State {
    uint8 operationState;
    mapping(uint8 => address) exchanges;
    address transferProxy;
    ITransferProxy transferProxyInstance;
    address vault;
    IVault vaultInstance;
    mapping(address => bool) validModules;
    mapping(address => bool) validFactories;
    mapping(address => bool) validSets;
    mapping(address => bool) disabledSets;
    address[] setTokens;
    mapping(address => bool) validPriceLibraries;
}
```

As the Solidity compiler does not perform optimizations as one might expect[6] it is necessary to ensure that structs and variables are aligned with 32 byte word boundaries. In the above struct the fields `uint8 operationState` and `address transferProxy` can be grouped together to save a full storage slot.

Fixed: SETPROTOCOL reordered the fields of the struct to optimize the use of storage slots.

[5] https://github.com/cgewecke/eth-gas-reporter
[6] https://solidity.readthedocs.io/en/latest/miscellaneous.html#layout-of-state-variables-in-storage

Silent overrides of mapping entries M ✓ Fixed

Function `addExchange` defined in contract `CoreInternal` allows the `owner` of the contract to add an exchange to the map of tracked exchanges via an 8-bit exchange identifier `uint8 _exchangeId`.

However, the function does not check whether an exchange with such id is already stored in the mapping and directly overrides the entry. Although the function is only callable by the `owner`, this makes it prone to errors when passing arguments to the function.

In such a scenario the system effectively removes, not adds, an exchange address without any direct indication of doing so[7]. For example when an exchanged is removed on purpose the appropriate event `ExchangeRemoved` is raised. This could have unintended consequences as it may temporarily block a certain exchange until it is added again, as any relevant operation first checks whether the exchange address is registered with the `state`.

Fixed: SETPROTOCOL added a `require` statement to ensure that the Exchange Id is not yet registered, this removes the possibility of silent overrides of existing entries.

Unused function parameter L ✓ Fixed

Function `settleRebalance` defined in library `StandardSettleRebalanceLibrary` accepts a parameter `address _manager` which is never used in the function body or stored. Hence, if not needed, it can be removed.

Fixed: SETPROTOCOL removed the unused parameter.

Function state mutability L ✓ Fixed

Several function throughput the codebase have their visibility set to `view` but since they are not even accessing contract data they should be further restricted to `pure`. These include:

- `validateTokenUsage()` defined in `ExchangeValidationLibrary`
- `calculateTokenFlows()` defined in `RebalancingHelperLibrary`

Other functions defined throughout the codebase can have their visibility restricted to `view` as they do not perform any state changing modifications:

- function `validateFillOrder` defined in `IssuanceOrderModule`
- function `getUnderlyingSetDetails` defined in `StandardSettleRebalanceLibrary`
- function `getUnderlyingSetsDetails` defined in `StandardStartRebalanceLibrary`
- function `calculateMinimumBid` defined in `StandardStartRebalanceLibrary`
- function `calculateCombinedUnitArrays` defined in `StandardStartRebalanceLibrary`
- function `calculateRebalancingSetIssueQuantity` defined in `PayableExchangeIssue`
- function `calculateAuctionPriceParameters` defined in `BTCETHRebalancingManager`

Fixed: SETPROTOCOL changed the function visibilities as recommended.

[7] The event issued when overriding an exchange address would be, misleadingly, `ExchangeAdded`

Unnecessary conversion L ✓ Fixed

Number literals can be implicitly converted to any integer type that is large enough to represent it without truncation[8] and no explicit conversion is necessary in this case. Occurrences of such conversions can be found throughout the `BTCETHRebalancingManager` contract:

```
uint256 naturalUnit;
...
naturalUnit = uint256(10**10);
...
naturalUnit = uint256(10**12);
...
uint256 componentUnitsInFullToken = _unit.mul(uint256(10**SET_TOKEN_DECIMALS))
    .div(_naturalUnit);
...
return _tokenPrice.mul(componentUnitsInFullToken).div(uint256(10**
    _tokenDecimals)).div(uint256(10**16));
```

Fixed: SETPROTOCOL removed all unnecessary conversions.

[8] https://solidity.readthedocs.io/en/latest/types.html#integer-types

Recommendations / Suggestions

- ☑ The error message in the fallback function of the `PayableExchangeIssue` contract contains a typo that can be fixed. Also a comment in the `LinearAuctionPriceCurve` misspells the variable name `priceDeonimator` and `multipled` in the `BTCETHRebalancingManager`.

- ☑ The `PayableExchangeIssue` contract has a comment-line declaring that **`public`** functions follow. However, following one **`public`** functions there are two **`private`** ones, so another corresponding comment line could be added or the top one removed.
A similar situation is in the `ExchangeIssueModule` contract, where `exchangeIssue()` is declared **`public`** but located in the external functions section.

- ☑ `RebalancedSet` tokens can be minted only while in `Default` state. This is enforced by two **`require`** statements executed in the `mint` function of the `RebalancingSetToken`. However, the error message for both statements is the same.

```
require(
    rebalanceState != RebalancingHelperLibrary.State.Rebalance,
    "RebalancingSetToken.mint: Cannot mint during Rebalance"
);

// Check that set is not in Drawdown State
require(
    rebalanceState != RebalancingHelperLibrary.State.Drawdown,
    "RebalancingSetToken.mint: Cannot mint during Rebalance"
);
```

- ☑ The event `BidPlaced` has the fields `bidder` and `quantity`. It might be convenient to index the bidders' addresses to simplify retrospective review and auditing.