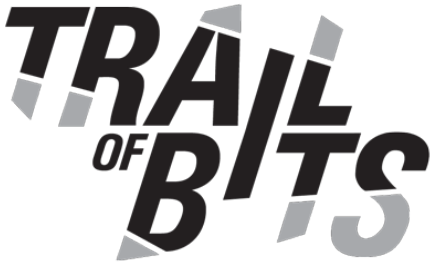

Set Protocol

Security Assessment

April 8th, 2019

Prepared For:
Felix Feng | *Set Protocol*
felix@setprotocol.com

Prepared By:
Robert Tonic | *Trail of Bits*
robert.tonic@trailofbits.com

Michael Colburn | *Trail of Bits*
michael.colburn@trailofbits.com

Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

JP Smith | *Trail of Bits*
jp@trailofbits.com

Changelog:
January 18, 2019: Initial report delivered to Set Protocol
January 24, 2019: Final report for publication
March 15, 2019: Report with additional week delivered to Set Protocol
March 29, 2019: Report updated to reflect new mitigations developed

Executive Summary

From January 7th through January 18th, Set Protocol engaged with Trail of Bits to review the security of the Set Protocol smart contracts. Trail of Bits conducted this review over the course of three person-weeks with three engineers working from d7ab276 in the set-protocol-contracts repository.

From March 11th through March 15th, Set Protocol re-engaged with Trail of Bits to review the security of the Set Protocol smart contracts. Trail of Bits conducted this review for one week with one engineer. Trail of Bits conducted this additional week of review from 0063f5e in the set-protocol-contracts repository.

From March 25th through March 29th, Trail of Bits reviewed fixes to issues discovered in the past two assessments for correctness. This review was conducted at commit b4acf14.

During the first review, Trail of Bits became familiar with the Set Protocol Solidity smart contracts and overall system design. The white paper was reviewed and compared to the implemented smart contracts to derive properties of the design and implementation. Subsequently, engineers tested these properties to verify their correctness and identify their implications in the system. Both manual and automated methods were used, including source analysis, property-based fuzzing, and symbolic execution.

In this first review, 11 findings emerged ranging from informational- to high-severity. Three issues related to the `SetToken` or `RebalancingSetToken` were identified, involving circular component references, rebalancing, and component upgrade fragility. Additionally, two issues related to ERC20 tokens involved approval race conditions and inline assembly usage. Investigations into deployment and migration processes identified two issues related to a lack of testing and contract configuration during deployment. Data validation issues were also identified that related to pricing libraries leading to potential denial of service for system components or users which perform price calculations, and whitelist usage potentially leading to incorrect execution restriction. Finally, one issue related to the decentralization of governance and one issue related to the 0x exchange wrapper's approval was identified.

In the second review, six additional flaws were identified ranging from informational- to high-severity. The most serious issue allowed an attacker to send an arbitrary number of orders with the same exchange identifier. Three medium-severity flaws were related to improper data validation in `RebalancingSetToken`, `ExchangeIssuanceModule` and `AddressArrayUtils` contracts. An additional finding allowed an attacker to manipulate the parameters of an event. The last issue related to potentially unsafe use of optimizations in Solidity.

In concluding our reviews, we noted that Set Protocol is a complex system. Numerous contracts with multiple tenancies comprise the system, compounded by third-party contract interactions. Despite the complexity, extensive unit testing is in place to test component functionality. Although thorough, we recommend expanding existing tests to include property testing and symbolic execution, using tools like Manticore and Echidna (see appendices D and E). Additionally, expected parameters for the Set Protocol should be designed and used to thoroughly test mathematical operations. The process for rebalancing should be reviewed in detail before adding any rebalancing sets to the rebalancing set component whitelist to prevent unexpected behavior.

Set Protocol should continue with further assessment of greater duration to allow deeper analysis. The system is operational according to the white paper and unit tests, however, unexplored paths, properties, and states may yield exploitable edge cases in a system of this complexity.

Retest Results

Trail of Bits performed a retest of the Set Protocol smart contracts from March 25 to 29 to verify the fixes to the issues reported during the two previous security reviews. Each of the issues was re-examined and verified by the audit team.

Emphasis was placed on investigating the code that was patched, the efficacy of the patches on the reported issues, and the security ramifications that may arise as a result of the code changes.

In total, Trail of Bits found that eleven issues were fully addressed, and six issues were not addressed. Issues that were not addressed include four medium severity issues, one issue of undetermined severity, and one informational issue.

	High	Medium	Undetermined	Info	TOTAL
Fixed	■■■■	■■■■■	■	■	7 issues
Partially Fixed					1 issue
Unfixed		■■■■	■	■	2 issues

Figure 1: Remediation status since the initial security assessment, as of March 29, 2019

Set Protocol has plans to address the remaining issues, or believes they are not a significant risk as the application is deployed or operated. Further information about the patching status of the findings and Set Protocol's response is in Appendix G.

Engagement Goals

The engagement sought to review the security and correctness of the Set Protocol smart contracts and their interactions, including the Truffle production migrations and configuration.

Specifically, we sought to answer the following questions:

- Is there any way the ERC20 tokens could be abused by users of the Set Protocol?
- Does the deployed system's operation match the white paper's description of operation?
- Is there third-party risk which the Set Protocol doesn't account for?
- Are there interactions between systems which should not be allowed?
- Is there any potential for a user to cause an unintended revert due to a Set Protocol system state?

To answer these questions, we performed detailed manual inspection of the contracts for known and unknown security flaws, extracted security properties from the Set Protocol whitepaper, and automated verification of certain properties.

System Properties

Trail of Bits reviewed a pre-publication edit to the Set Protocol whitepaper, dated January 7th, 2019, that defines numerous system properties. In our review, we abstracted 150 security properties from the whitepaper that are unique to Set Protocol. We then focused our review and verification efforts on groups of properties that were core to the protocol.

The components that received coverage are as follows:

- ERC20 Set and RebalancingSet tokens
- Set and RebalancingSet component operations
- Truffle migrations and contract configurations
- Core, Vault, and TransferProxy operations
- The 0x exchange wrapper

Manual Review

Set Protocol is complex. Some components require substantial initialization that hinders automated testing and verification. In these cases, Trail of Bits sought to manually review the code for adherence to the identified security properties.

During the first review, we focused on:

Time-locked governance. Set Protocol follows a time-locked governance protocol, with the intention to decentralize governance in the future. The concept of time-locked operations facilitates transparency of Set Protocol's on-chain governance.

ERC20 wrapper. The Set Protocol ERC20 wrapper handles non-compliant contract return values. Components of a Set conform to a standard ERC20 interface; however, some components don't conform to the standard ERC20 return values. To bypass this limitation, the Set Protocol uses an ERC20 wrapper to handle interactions with both compliant and non-compliant contracts using inline assembly. Manual analysis of the inline assembly used to parse the `returndata` of all third-party ERC20 interactions was performed to ensure correctness.

Truffle migrations. Project development and deployment leverages Truffle, a framework commonly used for building Solidity smart contracts. Truffle migrations yield a production deployment and configure the Set Protocol. Manual analysis revealed inadequate deployment testing and post-deployment contract configuration. The migrations have since been updated

In the second review, we focused on:

Failed Auction Settlement Procedure: The Set Protocol team introduced a new state called `Drawdown`, in the `RebalancingSetToken` contract. This new state was designed to mark rebalances that failed and cannot be completed or are under some sort of attack. Additionally, two new methods were added to allow for a transition into the `Drawdown` state, and to allow users to retrieve their collateral from the failed `RebalancingSetToken`. Manual analysis of these new changes was performed to ensure correctness.

Complementary contracts: A small set of complementary contracts that provide two useful features for final users: (1) sending Ether and atomically issuing a `RebalancingSetToken`, and (2) a set of contracts to *manage* `RebalancingSetToken` using different collateral such as Ether, DAI and Bitcoin. We reviewed every contract using manual analysis and identified only low-severity issues, which have since been remediated.

Custom deployment scripts: The truffle migration scripts were replaced by a custom script to deploy the contract in stages from the core libraries to the modules and high-level contracts. We manually reviewed the code used to deploy in the blockchain as well as every stage. Manual analysis revealed only minor flaws.

ExchangeIssuanceModule: This contract facilitates the issuance and redemption of `SetToken`s using exchange orders. It is a critical component of the Set Protocol system, since it parses, validates and executes orders from an array of bytes. We found through manual review two medium-severity issues potentially affecting this contract, both of which have been since remediated.

Automated Testing and Verification

Trail of Bits has developed three unique capabilities for testing smart contracts:

- Slither, a static analysis framework. Slither can statically verify algebraic relationships between Solidity variables. We used Slither to help identify a potentially erroneous dependence on `msg.sender`, described in Appendix F.
- Echidna, a smart contract fuzzer. Echidna can rapidly test security properties via malicious, coverage-guided test case generation. We used Echidna to test that ERC20 was correctly implemented, described in Appendix D.
- Manticore, a symbolic execution framework. Manticore can exhaustively test security properties via symbolic execution. We used Manticore to verify data validation in price calculations, described in Appendix E.

Automated testing techniques augment our manual security review, not replace it. Each technique has limitations: Slither may identify security properties that fail to hold when Solidity is compiled to EVM bytecode, Echidna may not randomly generate an edge case that violates a property, and Manticore may fail to complete its analysis. To mitigate these risks, we generate 10,000 test cases per property with Echidna and run Manticore to analysis completion when possible, then manually review all results.

Automated testing and verification was focused on the following system properties:

ERC20. Set Protocol contains several tokens which are intended to adhere to the ERC20 standard interface. Echidna was used to ensure the correctness of the `SetToken` implementation.

Property	Approach	Result
There is no way for the current user to increase its balance or decrease the balance of another user.	Echidna	Passed
The 0x0 address should not have a balance.	Echidna	Passed
A transfer of 0x0 should not be possible.	Echidna	Passed
The total supply of tokens should not change.	Echidna	Passed
A self-approval followed by self transfer-from should be possible.	Echidna	Passed
A self-approval and transfer-from to another account should be possible.	Echidna	Passed

A transfer of all a user's tokens plus one to another account should not be possible.	Echidna	Passed
A self-approval and transfer-from to 0x0 should not be possible.	Echidna	Passed

Price calculation. Price calculation libraries must return correct values. The Set Protocol system leverages a custom price calculation library across contracts. Manticore was used to symbolically verify the data validation and calculation functions used for pricing, producing values which lead to reverted transactions.

Property	Approach	Result
Validated parameters should not lead to a revert in price calculation.	Manticore	TOB-SP-008
Valid parameters for the price calculation function exist.	Manticore	Passed

Message sender. The message sender must be compared correctly to ensure appropriate authorization of certain contract actions. Using Slither, we extracted equality expressions involving the message sender, allowing more precise analysis of operator usage.

Property	Approach	Result
Message sender is appropriately used with exact equality operators.	Slither	Passed

Recommendations Summary

This section aggregates all the recommendations from the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Ensure all supported tokens using the ERC20 wrapper are tested and execute as expected.** The ERC20 wrapper uses inline assembly to parse return values from both compliant and non-compliant third-party ERC20 tokens. This is prone to both developer error and future unexpected behavior due to semantic opcode changes.

❑ **Ensure balances held by the Set Protocol can be migrated based on their upgrade strategy.** Due to the Vault's holding and tracking of user balances for various assets, ensure that there are strategies to migrate these values based on asset-upgrade methods, such as contract upgrade through value copy to a newly deployed contract.

❑ **Use the alternate increaseApproval and decreaseApproval functions in the OpenZeppelin ERC20 implementation when possible to avoid race conditions.** To ensure appropriate mitigation of the standard ERC20 approve race condition, consider leveraging the `increaseApproval` and `decreaseApproval` methods inherited from the OpenZeppelin implementation.

❑ **Consider using Echidna and Manticore to ensure no reverts are possible when calculating prices.** Reverts when calculating price information could lead to erroneous system states, or broken user-facing function behavior.

❑ **Publicly document who controls the address allowed to execute privileged functions.** Further documentation should detail what controls are in place to protect the corresponding private key(s) and in what scenarios administrator intervention will occur.

❑ **Assess the design of nested RebalancingSetTokens.** Nested components could rebalance in parallel with a parent token, causing unexpected behavior.

❑ **Consider disabling Solidity optimizations.** Measure the gas savings from optimizations, and carefully weigh that against the possibility of an optimization-related bug.

Long Term

❑ **Implement interfaces for non-conforming tokens to prevent use of inline assembly.** To avoid using inline assembly to parse non-conformant ERC20 token return values, consider implementing a contract to wrap these interactions. This will help prevent subtle errors which could arise from future changes in opcode semantics of the EVM.

❑ **Ensure Set component addresses are validated for expected functionality.** Ensuring each component of a Set operates as expected is important to ensure proper system operation.

❑ **Design and document a process to mitigate issues related to address changes in SetToken components.** Third-party contract upgrades of Set components resulting in a change of address may pose an issue with balance tracking in the Vault, and proper operation of a Set.

❑ **Require timeLockPeriod configuration before execution is allowed to occur on timeLockUpgrade protected methods.** While deployment configuration of contracts deriving from the `TimeLockUpgrade` contract is possible, requiring the `timeLockPeriod` being set at least once can help prevent unexpected execution of unconfigured contracts.

❑ **Implement testing for the Set Protocol deployment process.** Successful execution and post-deployment validity should be heavily tested.

❑ **Consider using a single approach to permissions across the Set Protocol.** Mixing approaches may result in unexpected restrictions.

❑ **Continue the development of the community-governance migration process.** This should be well defined and adhered to in order to prevent violations of trust during migration.

❑ **Consider redesigning the rebalancing process.** Deeply nested cyclic components of both Set and RebalancingSet could pose issues during a rebalance.

❑ **Continuously reassess the need for Solidity compiler optimizations.** Monitor the development and adoption of Solidity compiler optimizations to assess its maturity.

❑ **Carefully validate the values in the `LogPayableExchangeRedeem` event.** If manipulated parameters are detected, the transaction should be reverted.

Project Dashboard

Application Summary

Name	Set Protocol
Version	`D7ab276` (first review) `0063f5e` (second review) `B4acf14` (fix review)
Type	Solidity Smart Contracts
Platforms	Ethereum

Engagement Summary

Dates	January 7 - 18, March 11 - 15, 25 - 29
Method	Whitebox
Consultants Engaged	4
Level of Effort	5 person-weeks

Vulnerability Summary

Total High-Severity Issues	4	■■■■
Total Medium-Severity Issues	9	■■■■■■■■■
Total Informational-Severity Issues	2	■■
Total Undetermined-Severity Issues	2	■■
Total	17	

Category Breakdown

Patching	2	■■
Access Controls	1	■
Data Validation	7	■■■■■■■
Timing	2	■■
Denial of Service	3	■■■
Logging	1	■
Undefined Behavior	1	■
Total	17	

Findings Summary

#	Title	Type	Severity
1	Inline assembly is used to validate external contract calls	Data Validation	Medium
2	SetToken can reference itself as a component	Data Validation	Informational
3	SetToken components have limited upgradability	Patching	Medium
4	TimeLockUpgrade's timeLockPeriod remains default post-deployment	Timing	High
5	Race condition in the ERC20 approve function may lead to token theft	Timing	High
6	Deployments and migrations require further testing	Patching	High
7	Whitelist validations are not consistently used	Data Validation	Medium
8	Inadequate data validation in price libraries could result in unexpected reverts	Denial of Service	Medium
9	0x exchange wrapper is unable to increase approval for relay fees	Denial of Service	Medium
10	Current governance structure introduces counterparty risk	Access Controls	Informational
11	Component rebalance effectively pauses parent issuance	Denial of Service	Medium
12	Solidity compiler optimizations can be dangerous	Undefined Behavior	Undetermined
13	Insufficient validation of the rebalanceInterval parameter could produce a revert in the propose function	Data Validation	Medium

14	The ether quantity in the LogPayableExchangeRedeem event cannot be trusted	Logging	Undetermined
15	Insufficient input validation in ExchangeIssuanceModule functions	Data Validation	Medium
16	hasDuplicate runs out of gas when the input list is empty	Data Validation	Medium
17	executeExchangeOrders fails to properly validate repeated exchanges	Data Validation	High

1. Inline assembly is used to validate external contract calls

Severity: Medium
Type: Data Validation
Target: `contracts/lib/ERC20Wrapper.sol`

Difficulty: High
Finding ID: TOB-SP-001

Description
In the `ERC20Wrapper` library, a `checkSuccess` function is defined and used to validate external function calls to an arbitrary address implementing the standard `ERC20` interface. The `checkSuccess` function uses inline assembly to test the `returndata` values of the last function call. It must be executed directly after each function call, which requires validation.

```
function checkSuccess(
)
    private
    pure
    returns (bool)
{
    // default to failure
    uint256 returnValue = 0;

    assembly {
        // check number of bytes returned from last function call
        switch returndatasize

        // no bytes returned: assume success
        case 0x0 {
            returnValue := 1
        }

        // 32 bytes returned
        case 0x20 {
            // copy 32 bytes into scratch space
            returndatacopy(0x0, 0x0, 0x20)

            // load those bytes into returnValue
            returnValue := mload(0x0)
        }

        // not sure what was returned: dont mark as success
        default { }
    }

    // check if returned value is one or nothing
    return returnValue == 1;
}
```

Figure 1: The `checkSuccess` *function definition*

```
function transfer(
    address _token,
    address _to,
    uint256 _quantity
)
    external
{
    IERC20(_token).transfer(_to, _quantity);

    // Check that transfer returns true or null
    require(
        checkSuccess(),
        "ERC20Wrapper.transfer: Bad return value"
    );
}
```

Figure 2: Example usage of `checkSuccess`

The use of inline assembly in this fashion is prone to compatibility issues in future releases of Solidity, and could be subject to further unexpected edge cases. Additionally, developer error could lead to the introduction of bugs with the use of `checkSuccess`, since it is sensitive to the order of execution.

See Appendix C for further discussion regarding the use of inline assembly usage.

Exploit Scenarios
The Set Protocol system changes the version of Solidity to a newer version. This version has breaking changes surrounding `returndata`. Subsequently, this leads to a broken `checkSuccess`, leading to unintended return values from calls using the `ERC20Wrapper` library.

A new feature requires further validation of return values in the `ERC20Wrapper` library. A developer adds this validation, but fails to maintain the order of execution between the external call and the `checkSuccess` function. An invalid calculation could occur in `checkSuccess`.

Recommendation
Short term, test all supported tokens using this wrapper to ensure they execute as expected.

Long term, implement interfaces for non-conforming tokens to prevent use of inline assembly.

2. SetToken can reference itself as a component

Severity: Informational | Difficulty: Low
Type: Data Validation | Finding ID: TOB-SP-002
Target: `contracts/core/tokens/{Rebalancing}SetToken.sol`

Description
Due to `SetToken`'s implementation of `ERC20` interface methods, a `SetToken` can be included as a component of another `SetToken`. While this is expected behavior, a `SetToken` may be included as a component of itself due to the predictable nature of Ethereum addresses.

Due to Ethereum contract addresses deriving from the creator's address and nonce, it is reasonable to assume that a newly issued `SetToken` can be initialized with its own address included as a component address. Subsequently, this would result in a self-referencing Set which is valid by system design according to the white paper.

Exploit Scenario
A `SetToken` is issued with its own address provided as a component address during contract construction. This corner case could cause some of the Set Protocol components to behave in an unexpected way.

Recommendation
Short term, consider validating each component address to ensure it is not the same as the `SetToken` address.

Long term, ensure addresses that are called are validated for functionality.

3. SetToken components have limited upgradability

Severity: Medium
Type: Patching
Target: `contracts/core/tokens/SetToken.sol`

Difficulty: Low
Finding ID: TOB-SP-003

Description
When `SetToken`s are issued, component addresses are provided during the construction of the `SetToken`. After a `SetToken` is issued, the component addresses cannot be changed. Because of this, a balance could be rendered unclaimable if a component of a Set is upgraded in a way that orphans the original address.

```
constructor(
    address _factory,
    address[] _components,
    uint256[] _units,
    uint256 _naturalUnit,
    string _name,
    string _symbol
)
...
{
   // Add component data to components and units state variables
   components.push(currentComponent);
   ...

}
```

Figure 1: The `SetToken` *constructor setting the component addresses*

Exploit Scenario
Bob deploys an `ERC20`-compliant `MaliciousToken`. Alice creates a `SetToken` using `MaliciousToken` as a component. Bob subsequently triggers an upgrade to `MaliciousToken`, pausing the component-tracked address and migrating values to a new address. Alice subsequently cannot continue using the provisioned `SetToken` since the component address of `MaliciousToken` is no longer valid.

Recommendation
Short term, ensure balances can be migrated for each token based on their upgrade method.

Long term, design and document a process to mitigate issues caused by address-related changes in `SetToken` components.

4. TimeLockUpgrade's timeLockPeriod remains default post-deployment

Severity: High
Type: Timing
Target: `contracts/lib/TimeLockUpgrade.sol`

Difficulty: Low
Finding ID: TOB-SP-004

Description
Several contracts inherit functionality from the `TimeLockUpgrade` contract, and use its `timeLockUpgrade` modifier. However, the `setTimeLock` function is never invoked after a contract is deployed by `2_core.js`, resulting in a default `timeLockPeriod` value of `0` and all methods using the `timeLockUpgrade` modifier being invokable at any time.

This `timeLockUpgrade` modifier bypass allows for owners of the system to introduce new Modules, Price Libraries, and similar without waiting, as the white paper describes in Figure 1.

> [...]Our intention is to create a system that is as decentralized and trustless as possible. See below for the limited capabilities of the governors:
>
> - Add and remove Modules, ExchangeWrappers, Signature Validator, Price Libraries, and Factories. Each addition is a Time-Locked operation, requiring a 7 or 14 day period before they can become operational
>
> [...]

Figure 1: The excerpt from the white paper describing expected functionality of the time lock

```
modifier timeLockUpgrade() {
    // If the time lock period is 0, then allow non-timebound upgrades.
    // This is useful for initialization of the protocol and for testing.
    if (timeLockPeriod == 0) {
        _;

        return;
    }
    ...
}
```

Figure 2: The passthrough if `timeLockPeriod` *is* `0` *in the* `timeLockUpgrade` *modifier*

```
function addFactory(
    address _factory
)
    external
    onlyOwner
    timeLockUpgrade
```

```
{
    state.validFactories[_factory] = true;

    emit FactoryAdded(
        _factory
    );
}
```

Figure 3: An example `timeLockUpgrade` *protected* `CoreInternal` *contract method*

```
async function deployCoreContracts(deployer, network) {
  ...
  // Deploy Core
  await deployer.deploy(Core, TransferProxy.address, Vault.address,
SignatureValidator.address);
  …
}
```

Figure 4: The `2_core.js` *migration which does not use* `setTimeLock` *on the Core contract, which inherits from Figure 3's* `CoreInternal` *contract*

Exploit Scenario

The Set Protocol system is deployed. After deployment, the `setTimeLock` is never invoked to set a `timeLockPeriod`. Arbitrary system components are registered, despite a time lock period defined by the white paper.

Recommendation

Short term, configure a `timeLockPeriod` in the `Core` system deployment.

Long term, require `timeLockPeriod` configuration before execution is allowed to occur on methods protected by `timeLockUpgrade`.

5. Race condition in the ERC20 approve function may lead to token theft

Severity: Medium
Difficulty: High
Type: Timing
Finding ID: TOB-SP-005
Target: `ERC20 Tokens`

Description
A known race condition in the ERC20 standard, on the `approve` function, could lead to the theft of tokens.

The ERC20 standard describes how to create generic token contracts. Among others, an ERC20 contract defines these two functions:

- `transferFrom(from, to, value)`
- `approve(spender, value)`

These functions give permission to a third party to spend tokens. Once the function `approve(spender, value)` has been called by a user, `spender` can spend up to `value` of the user's tokens by calling `transferFrom(user, to, value).`

This schema is vulnerable to a race condition when the user calls `approve` a second time on a `spender` that has already been allowed. If the spender sees the transaction containing the call before it has been mined, then the spender can call `transferFrom` to transfer the previous value and still receive the authorization to transfer the new value.

Exploit Scenario
1. Alice calls `approve(Bob, 1000)`. This allows Bob to spend 1000 tokens.
2. Alice changes her mind and calls `approve(Bob, 500).` Once mined, this will decrease to 500 the number of tokens that Bob can spend.
3. Bob sees Alice's second transaction and calls `transferFrom(Alice, X, 1000)` before `approve(Bob, 500)` has been mined.
4. If Bob's transaction is mined before Alice's, 1000 tokens will be transferred by Bob. Once Alice's transaction is mined, Bob can call `transferFrom(Alice, X, 500)`. Bob has transferred 1500 tokens, contrary to Alice's intention.

Recommendations
While this issue is known and can have a severe impact, there is no straightforward solution.

One workaround is to use two non-ERC20 functions allowing a user to increase and decrease the approve (see `increaseApproval` and `decreaseApproval` of StandardToken.sol#L63-L98).

Another workaround is to forbid a call to `approve` if all the previous tokens are not spent by adding a `require` to approve. This prevents the race condition but it may result in unexpected behavior for a third party.

```
require(_approvals[msg.sender][guy] == 0)
```

This issue is a flaw in the ERC20 design. It cannot be fixed without modifications to the standard. It must be considered by developers while writing code.

6. Deployments and migrations require further testing

Severity: High
Type: Patching
Target: `Truffle Migration`

Difficulty: Low
Finding ID: TOB-SP-006

Description
During the engagement, the Set Protocol Truffle migrations were tested for functionality. This resulted in errors regarding a newly added whitelist contract which was a dependency for another contract.

These errors indicate a need for further testing to ensure deployments and their post-deployment configurations are successful and adhere to the white paper. Failure to do so could result in an erroneous production deployment. Testing should also be expanded to account for 3rd party system interactions. Systems such as exchanges should be simulated to ensure adequate testing locally.

Exploit Scenario
The Set Protocol migrations are executed. All but one contract deployed successfully. The failed contract puts the other system components into an invalid state, requiring a redeployment.

Recommendation
Short term, ensure Truffle migrations result in a production-ready configuration programmatically, with the least human interaction necessary to verify post-deployment state and correctness.

Long term, implement testing for the deployment process to ensure successful execution and post-deployment validity.

7. Whitelist validations are not consistently used

Severity: Medium | Difficulty: Low
Type: Data Validation | Finding ID: TOB-SP-007
Target: `contracts/core/tokens/rebalancing-libraries/StandardProposeLibrary.sol`, `contracts/core/tokens/RebalancingSetToken.sol`

Description
Managers can start proposals for a `RebalancingSetToken`. During the proposal period, new components can be proposed for the token. However, when validating a newly proposed component, validation occurs in several places without clear purpose. This could lead to permissions issues when permissions are intended to be managed by whitelists, but are in fact hard coded within a function's definition.

```
// Validate proposal inputs and initialize auctionParameters
auctionParameters = StandardProposeLibrary.propose(
    _nextSet,
    ...
    componentWhiteListInstance,
    ...
);
```

Figure 1: The `RebalancingSetToken`*'s call to the* `StandardProposeLibrary.propose` *function with its* `componentWhiteListInstance` *as an argument*

```
function propose(
    address _nextSet,
    address _auctionLibrary,
    uint256 _auctionTimeToPivot,
    uint256 _auctionStartPrice,
    uint256 _auctionPivotPrice,
    IWhiteList _componentWhiteList,
    ProposeAuctionParameters memory _proposeParameters
)
    internal
    returns (RebalancingHelperLibrary.AuctionPriceParameters)
{
  ...
  // Check that new proposed Set is valid Set created by Core
  require(
      _proposeParameters.coreInstance.validSets(_nextSet),
      "RebalancingSetToken.propose: Invalid or disabled proposed SetToken
address"
  );

  // Check proposed components on whitelist. This is to ensure managers are
```

```
unable to add contract addresses
  // to a propose that prohibit the set from carrying out an auction i.e. a
token that only the manager possesses
  require(

_componentWhiteList.areValidAddresses(ISetToken(_nextSet).getComponents()),
      "RebalancingSetToken.propose: Proposed set contains invalid component
token"
  );
  ...
}
```

Figure 2: The `StandardProposeLibrary`.propose *function, validating a set through both* `Core` *and the provided* `componentWhitelist`

Exploit Scenario

The Set Protocol system is successfully deployed. Upon deployment and creation of a `RebalancingSetToken`, a whitelist is used to attempt to restrict execution. Because the `Core` invalidates the `Set`, whitelist validation of the `Set` is impossible.

Recommendation

Short term, consistently use validations in proposals. Variation between whitelist and in-method validation could lead to developer error or improper configuration.

Long term, consider using a single approach to validations across the Set Protocol.

8. Inadequate data validation in price libraries could result in unexpected reverts

Severity: Medium | Difficulty: Low
Type: Denial of Service | Finding ID: TOB-SP-008
Target: `core/lib/auction-price-libraries/LinearAuctionPriceCurve.sol`

Description
The validation of parameters in the price libraries is insufficient. The parameters for the auction price can be validated using `validateAuctionPriceParameters` as shown in Figure 1.

```
function validateAuctionPriceParameters(
        RebalancingHelperLibrary.AuctionPriceParameters _auctionParameters
)
    public
    view
{
    // Require pivot price to be greater than 0.5 * price denominator
    // Equivalent to oldSet/newSet = 0.5
    require(
      _auctionParameters.auctionPivotPrice >
priceDenominator.div(MIN_PIVOT_PRICE_DIVISOR),
        "LinearAuctionPriceCurve.validateAuctionPriceParameters: Pivot
price too low"
    );
    // Require pivot price to be less than 5 * price denominator
    // Equivalent to oldSet/newSet = 5
    require(
        _auctionParameters.auctionPivotPrice <
priceDenominator.mul(MAX_PIVOT_PRICE_NUMERATOR),
        "LinearAuctionPriceCurve.validateAuctionPriceParameters: Pivot
price too high"
    );
}
```

Figure 1: The `validateAuctionPriceParameters` *function definition*

Figure 2 shows how users and contracts can employ the `getCurrentPrice` function to obtain the price of certain auctions given the function's parameters.

```
/*
     * Calculate the current priceRatio for an auction given defined price
and time parameters
     *
```

```
     * @param _auctionPriceParameters     Struct containing relevant
auction price parameters
     * @return uint256                    The auction price numerator
     * @return uint256                    The auction price denominator
*/
function getCurrentPrice(
    RebalancingHelperLibrary.AuctionPriceParameters _auctionParameters
)
    public
    view
    returns (uint256, uint256)
{
    // Calculate how much time has elapsed since start of auction
   uint256 elapsed =
block.timestamp.sub(_auctionParameters.auctionStartTime);

    // Initialize numerator and denominator
    uint256 priceNumerator = _auctionParameters.auctionPivotPrice;
    uint256 currentPriceDenominator = priceDenominator;
    ...
```

Figure 2: The `getCurrentPrice` *function declaration*

However, if an auction price is created with certain invalid parameters, a call to obtain its price using `getCurrentPrice` will cause a revert, blocking the proposal. For instance, the following parameters are considered valid by `validateAuctionPriceParameters`, but it will cause `getCurrentPrice` to revert:

- `auctionStartTime = 597123632108438120153802479587592840174376045019910281344224284188585240821 76`
  `auctionTimeToPivot = 203470107864034093222171347241117752212257985717009166149069579105694111498 24`
  `auctionStartPrice = 0`
  `auctionPivotPrice = 2048`
  `block.number = 868440669281978740676300365494396350252278809743161901176112729130034161254 41`

These values are just an example, as there are numerous parameters that can trigger this issue.

Exploit Scenario
Alice submits a proposal using some parameters that causes a computation during a call to `getCurrentPrice` to revert. As a result, she is unable to obtain the price of a proposal.

Recommendation
Short term, carefully validate the parameters of the proposal. Revert if they are not valid.

Long term, consider using the Echidna fuzzer or the Manticore symbolic executor to check that no revert can happen during the call to `getCurrentPrice`.

9. 0x exchange wrapper cannot increase approval for relay fees

Severity: Medium | Difficulty: Low
Type: Denial of Service | Finding ID: TOB-SP-009
Target: `contracts/core/exchange-wrappers/ZeroExExchangeWrapper.sol`

Description
The `ZeroExExchangeWrapper` constructor approves of the `_zeroExProxy` address to transfer the `_zeroExToken` on behalf of the `ZeroExExchangeWrapper` contract, allowing balance transfers in the event of 0x exchange order relay fees. However, there is no method to later increase this approval, resulting in a finite amount of transfers which could eventually be depleted by system use and result in trapped funds.

```
/**
 * Initialize exchange wrapper with required addresses to facilitate 0x
orders
 *
 * @param _core                 Deployed Core contract
 * @param _zeroExExchange       0x Exchange contract for filling orders
 * @param _zeroExProxy          0x Proxy contract for transferring
 * @param _zeroExToken          ZRX token contract addressed used for 0x
relayer fees
 * @param _setTransferProxy     Set Protocol transfer proxy contract
 */
constructor(
    address _core,
    address _zeroExExchange,
    address _zeroExProxy,
    address _zeroExToken,
    address _setTransferProxy
)
    public
{
    core = _core;
    zeroExExchange = _zeroExExchange;
    zeroExProxy = _zeroExProxy;
    zeroExToken = _zeroExToken;
    setTransferProxy = _setTransferProxy;

    // Approve transfer of 0x token from this wrapper in the event of
zeroExOrder relayer fees
    ERC20.approve(
        _zeroExToken,
        _zeroExProxy,
        CommonMath.maxUInt256()
    );
```

```
}
```

Figure 1: The constructor of the `ZeroExExchangeWrapper` *contract*

Exploit Scenario
The `ZeroExExchangeWrapper` contract is deployed successfully. Over time, the contract's approval is depleted through use. No further transfers are possible due to an inability to increase approval.

Recommendation
Short term, ensure there is a method to increase the approval of the 0x exchange wrapper. Without this, funds may become trapped.

Long term, care should be taken to ensure balances of the 0x exchange wrapper can be appropriately managed.

10. Current governance structure introduces counterparty risk

Severity: Informational
Type: Access Controls
Target: `Set Protocol Governance`

Difficulty: Low
Finding ID: TOB-SP-010

Description
While the Set Protocol team eventually plans to move to a community-governance model, currently all privileged activities are carried out by the Set Protocol team. These actions include the shutdown and restart of the `Core` contract, enabling and disabling individual `Sets` in `Core`, and managing the component whitelist for `RebalancingSetTokenFactory`. Participants in the Set Protocol community are implicitly trusting the Set Protocol team to act in the community's best interest.

Exploit Scenario
The Set Protocol team deems it necessary to disable a misbehaving `Set`. The community is divided as to whether this was the correct course of action, damaging Set Protocol's reputation.

Recommendation
Short term, publicly document who controls the address that can execute privileged functions, what controls are in place to protect the corresponding private key(s) and in what scenarios administrator intervention will occur.

Long term, continue to develop a plan to migrate toward a community-governance model. This should be well defined and adhered to in order to prevent violations of trust during migration.

11. Component rebalance effectively pauses parent issuance

Severity: Medium
Type: Denial of Service
Target: `Sets`

Difficulty: Low
Finding ID: TOB-SP-011

Description
Since Sets, including Rebalancing Sets, conform to the `ERC20` specification, they can contain other Sets as components. Issuance of parent Sets in this case will rely on the issuance of child Sets. However, during a rebalance issuance is paused. This potentially could lead to liquidity problems when a component Rebalancing Set goes through a rebalance while the parent Rebalancing Set is also going through a rebalance (even if separated by multiple levels of nested Sets).

Exploit Scenario
The manager of a `RebalancingSetToken` (Token A) issues a proposal to rebalance to a new Set containing as one of its components another `RebalancingSetToken` (Token B). Shortly thereafter, the manager of Token B issues a proposal which transitions to the `Rebalance` state before Token A's rebalancing occurs. When Token A enters the `Rebalance` state, no new Token B may be issued due to issuance being restricted during a Rebalance. This reduces the liquidity pool and potentially allows current Token B holders to acquire Token A's `currentSet` at a discount through the auction mechanism.

Recommendation
Short term, assess the design of nested `RebalancingSetTokens` through components. Architect the rebalancing process to account for nested components which could rebalance in parallel.

Long term, consider redesigning the rebalancing process to account for deeply nested cyclic components during a rebalance.

12. Solidity compiler optimizations can be dangerous

Severity: Undetermined
Type: Undefined Behavior
Target: `truffle.js`

Difficulty: Low
Finding ID: TOB-SP-012

Description
Set Protocol has enabled optional compiler optimizations in Solidity.

There have been several bugs with security implications related to optimizations. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default. It is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the emscripten-generated `solc-js` compiler used by Truffle and Remix persisted until just a few months ago. The fix for this bug was not reported in the Solidity CHANGELOG.

A compiler audit of Solidity from November, 2018 concluded that the optional optimizations may not be safe. Moreover, the Common Subexpression Elimination (CSE) optimization procedure is "implemented in a very fragile manner, with manual access to indexes, multiple structures with almost identical behavior, and up to four levels of conditional nesting in the same function." Similar code in other large projects has resulted in bugs.

There are likely latent bugs related to optimization, and/or new bugs that will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the Set Protocol contracts.

Recommendation
Short term, measure the gas savings from optimizations, and carefully weigh that against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess its maturity.

13. Insufficient validation of the rebalanceInterval parameter could produce a revert in the propose function

Severity: Medium | Difficulty: Medium
Type: Data Validation | Finding ID: TOB-SP-013
Target: `StandardProposeLibrary.sol, RebalancingSetToken.sol`

Description
The `propose` function could unexpectedly revert if a `RebalancingSetToken` is created with an unexpectedly large `rebalanceInterval` parameter.

Before creating a new `RebalancingSetToken`, its parameters are validated in its constructor function. Later, the `propose` function in `StandardProposeLibrary`, as shown in Figure 1, uses such parameters for validation checks on the `RebalancingToken` before starting a rebalance.

```
   function propose(
        address _nextSet,
        address _auctionLibrary,
        uint256 _auctionTimeToPivot,
        uint256 _auctionStartPrice,
        uint256 _auctionPivotPrice,
        address _factoryAddress,
        address _componentWhiteListAddress,
        ProposeAuctionParameters memory _proposeParameters
    )
        public
        returns (RebalancingHelperLibrary.AuctionPriceParameters memory)
    {
        ...

        // Make sure enough time has passed from last rebalance to start a
new proposal
        require(
            block.timestamp >=
_proposeParameters.lastRebalanceTimestamp.add(
                _proposeParameters.rebalanceInterval
            ),
            "RebalancingSetToken.propose: Rebalance interval not elapsed"
        );
        ...
```

Figure 1: The `propose` *function declaration*

However, the verification that the rebalance is not happening too frequently can revert if the `RebalancingSetToken` was created using a very large `rebalanceInterval` parameter.

Exploit Scenario
A `RebalancingSetToken` is created using a very large `rebalanceInterval.` As a result, it cannot be rebalanced at all and it will be stuck in the `Default` state.

Recommendation
Short term, carefully validate the parameters of the `RebalancingSetToken` creation. Revert if they are not valid.

Long term, consider using the Echidna fuzzer or the Manticore symbolic executor to check that no revert can happen during the call to `propose`.

14. The ether quantity in the `LogPayableExchangeRedeem` event cannot be trusted

Severity: Undetermined
Type: Logging
Target: `PayableExchangeIssuance.sol`

Difficulty: High
Finding ID: TOB-SP-014

Description
The ether quantity in the `LogPayableExchangeRedeem` event can be manipulated in the context of a rebalancing set redemption into a wrapped ether token.
The `LogPayableExchangeRedeem` event shown in Figure 1, contains the `etherQuantity` parameter.

```
    event LogPayableExchangeRedeem(
        address setAddress,
        address indexed callerAddress,
        uint256 etherQuantity
    );
```

Figure 1: The `LogPayableExchangeRedeem` *event declaration*

This event logs the amount of ether redeemed when a user calls the `redeemRebalancingSetIntoEther` function as shown in Figure 2.

```
    function redeemRebalancingSetIntoEther(
        address _rebalancingSetAddress,
        uint256 _rebalancingSetQuantity,
        ExchangeIssuanceLibrary.ExchangeIssuanceParams memory
_exchangeIssuanceParams,
        bytes memory _orderData
    )
        public
        nonReentrant
    {
        ...
        // Withdraw eth from WETH
        uint256 wethBalance = ERC20Wrapper.balanceOf(
            weth,
            address(this)
        );
        wethInstance.withdraw(wethBalance);

        // Send eth to user
        msg.sender.transfer(wethBalance);
```

```
        emit LogPayableExchangeRedeem(
            _rebalancingSetAddress,
            msg.sender,
            wethBalance
        );
    }
```

Figure 2: The `redeemRebalancingSetIntoEther` *function declaration*

The `etherQuantity` is read directly from the balance of the corresponding WETH token used by `PayableExchangeIssuance`. Therefore, it can be increased by depositing ether into such an address before calling the `redeemRebalancingSetIntoEther` function.

Exploit Scenario
Bob deposits ether into WETH token used by `PayableExchangeIssuance` before calling the `redeemRebalancingSetIntoEther` function. When this function is called, the emitted LogPayableExchangeRedeem event could be used by off-chain code to compute some important values (e.g. the amount of ether that Bob has in the system) potentially causing some unexpected behavior (e.g integer underflow).

Recommendation
In the short term, review your off-chain code to make sure it cannot be manipulated using a `LogPayableExchangeRedeem` event.

In the long term, carefully validate the values in the `LogPayableExchangeRedeem` to avoid manipulation by malicious users.

15. Insufficient input validation in ExchangeIssuanceModule functions

Severity: Medium
Type: Data Validation
Target: `ExchangeIssuanceModule.sol`

Difficulty: Low
Finding ID: TOB-SP-015

Description
The functions in the `ExchangeIssuanceModule` contract fail to properly validate their parameters.

The `exchangeIssue` and `exchangeRedeem` functions defined in `ExchangeIssuanceModule` as shown in Figures 1 and Figure 2 are directly used by unauthenticated users to perform trades and redeem sets via exchange wrappers.

```
   /**
     * Performs trades via exchange wrappers to acquire components and
issues a Set to the caller
     *
     * @param _exchangeIssuanceParams              A Struct containing
exchange issuance metadata
     * @param _orderData                           Bytes array containing
the exchange orders to execute
     */
    function exchangeIssue(
        ExchangeIssuanceLibrary.ExchangeIssuanceParams memory
_exchangeIssuanceParams,
        bytes memory _orderData
    )
        public
        nonReentrant
    {
        // Ensures validity of exchangeIssuanceParams
        validateExchangeIssuanceParams(_exchangeIssuanceParams);
        ...
```

Figure 1: The `exchangeIssue` *function declaration*

```
   /**
     * Redeems a Set and performs trades via exchange wrappers for
specified receive tokens. The receive
     * tokens are attributed to the caller.
     *
     * @param _exchangeIssuanceParams              A Struct containing
exchange issuance metadata
     * @param _orderData                           Bytes array containing
the exchange orders to execute
```

```
     */
    function exchangeRedeem(
        ExchangeIssuanceLibrary.ExchangeIssuanceParams memory
_exchangeIssuanceParams,
        bytes memory _orderData
    )
        public
        nonReentrant
    {
        // Validate exchangeIssuanceParams
        validateExchangeIssuanceParams(_exchangeIssuanceParams);
        ...
```

Figure 2: The `exchangeRedeem` *function declaration*

They validate the `_exchangeIssuanceParams` parameter using the `validateExchangeIssuanceParams` function which calls the `validateSendTokenParams` (shown in Figure 3).

```
   /**
     * Validates that the send tokens inputs are valid
     *
     * @param _core                             The address of Core
     * @param _sendTokenExchangeIds             List of exchange wrapper
enumerations corresponding to
     *                                              the wrapper that will
handle the component
     * @param _sendTokens                       The address of the send tokens
     * @param _sendTokenAmounts                 The quantities of send tokens
     */
    function validateSendTokenParams(
        address _core,
        uint8[] memory _sendTokenExchangeIds,
        address[] memory _sendTokens,
        uint256[] memory _sendTokenAmounts
    )
        internal
        view
    {
        require(
            _sendTokenExchangeIds.length == _sendTokens.length &&
            _sendTokens.length == _sendTokenAmounts.length,
            "ExchangeIssuanceLibrary.validateSendTokenParams: Send token
inputs must be of the same length"
        );

        for (uint256 i = 0; i < _sendTokenExchangeIds.length; i++) {
```

```
            // Make sure all exchanges are valid
            require(
                ICore(_core).exchangeIds(_sendTokenExchangeIds[i]) !=
address(0),
                "ExchangeIssuanceLibrary.validateSendTokenParams: Must be
valid exchange"
            );

            // Make sure all send token amounts are non-zero
            require(
                _sendTokenAmounts[i] > 0,
                "ExchangeIssuanceLibrary.validateSendTokenParams: Send
amounts must be positive"
            );
        }
    }
```

Figure 3: The complete `validateSendTokenParams` *function*

However, the validation fails to detect when the list of tokens and amounts are empty or contain duplicates.

Exploit Scenario
A user calls the `exchangeIssue` or `exchangeRedeem` function using a list of `sendToken` with repeated tokens. This operation breaks an important invariant in the contracts, potentially causing unexpected behavior in other components.

Recommendation
Short term, carefully validate the parameters of the `ExchangeIssuanceModule` functions. Revert if they are not valid.

Long term, consider using the Echidna fuzzer or the Manticore symbolic executor to check that invalid parameters are properly detected.

16. `hasDuplicate` runs out of gas when the input list is empty

Severity: Medium
Type: Data Validation
Target: `AddressArrayUtils.sol`

Difficulty: Undetermined
Finding ID: TOB-SP-016

Description
The `hasDuplicate` function, which determines if a list of addresses contains duplicates, is incorrectly implemented.
The hasDuplicate function is shown in Figure 1. Its documentation states that it returns true if it finds duplicates and false otherwise.

```
    /**
     * Returns whether or not there's a duplicate. Runs in O(n^2).
     * @param A Array to search
     * @return Returns true if duplicate, false otherwise
     */
    function hasDuplicate(address[] memory A) internal pure returns (bool)
{
        for (uint256 i = 0; i < A.length - 1; i++) {
            for (uint256 j = i + 1; j < A.length; j++) {
                if (A[i] == A[j]) {
                    return true;
                }
            }
        }
        return false;
    }
```

Figure 1: The complete `hasDuplicate` *function*

However, this function has a flaw: if it is called using an empty dynamic array, it will trigger an unsigned integer underflow when calculating the loop bound (`A.length - 1`), causing it to loop until it runs out of gas.

Exploit Scenario
The Set Protocol team uses the `hasDuplicate` function elsewhere in the system, introducing a potential security (e.g., denial of service) or correctness issue.

Recommendation
Short term, fix the implementation of `hasDuplicate` to return the correct value when the input list is empty.

Long term, consider using the Echidna fuzzer or the Manticore symbolic executor to check the correctness of the `hasDuplicate` function.

17. `executeExchangeOrders` fails to properly validate repeated exchanges

Severity: High
Type: Data Validation
Target: `ExchangeExecution.sol`

Difficulty: High
Finding ID: TOB-SP-016

Description
The `executeExchangeOrders` function fails to properly validate repeated exchanges when it parses orders.

The `executeExchangeOrders` function is shown in Figure 1. This function parses, validates and executes the exchange orders. One important validation is the detection of repeated exchanges in the list of orders, performed using the & operator between `exchangeBitIndex` and `calledExchanges`.

```
   /**
     * Execute the exchange orders by parsing the order data and
facilitating the transfers. Each
     * header represents a batch of orders for a particular exchange (0x,
Kyber)
     *
     * @param _orderData               Bytes array containing the exchange
orders to execute
     */
    function executeExchangeOrders(
        bytes memory _orderData
    )
        internal
    {
        // Bitmask integer of called exchanges. Acts as a lock so that
duplicate exchange headers are not passed in.
        uint256 calledExchanges = 0;
        uint256 scannedBytes = 0;
        while (scannedBytes < _orderData.length) {
            ...
            // Verify exchange has not already been called
            uint256 exchangeBitIndex = 2 ** header.exchange;
            require(
                (calledExchanges & exchangeBitIndex) == 0,
                "ExchangeExecution.executeExchangeOrders: Exchange already
called"
            );

            ...
```

```
            // Update scanned bytes with header and body lengths
            scannedBytes = scannedBytes.add(exchangeDataLength);

            // Increment bit of current exchange to ensure non-duplicate
entries
            calledExchanges = calledExchanges.add(exchangeBitIndex);
        }
    }
```

Figure 1: Part of the `executeExchangeOrders` *function*

However, this function triggers an integer overflow in the computation of `exchangeBitIndex`: despite this variable is declared as `uint256`, the computation is performed using only `uint8` (2 and `header.exchange`). Therefore, any exchange identifier larger or equal than 8 will overflow. Moreover, using `header.exchange == 8`, will cause `exchangeBitIndex` and `calledExchanges` to be zero, allowing an attacker to bypass the repeated exchange verification.

Exploit Scenario
If the exchange identifier number 8 is valid, an attacker can create an arbitrary amount of orders using such identifier to bypass the check in `executeExchangeOrders`. This operation breaks an important invariant in the contracts, potentially causing unexpected behavior in other components.

Recommendation
Short term, reimplement the detection of repeated exchanges in the list of orders without using arithmetic functions like exponentiation, which are prone to integer-overflow issues.

Long term, consider using the Echidna fuzzer or the Manticore symbolic executor to check the correctness of the `hasDuplicate` function.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal

	implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

B. Code Quality

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

General

- Instead of restricting execution to particular addresses through the use of `require` in contract method bodies, consider moving them into function modifiers for re-use across the codebase.

Deployment scripts:

- The `3_modules.ts` deployment script contains the function `deployRebalancingAuctionModule()` that deploys the `RebalanceAuctionModule` contract. It is recommended to rename such functions to match the name of the contract it deploys to make the code easier to understand and review.
- The `3_modules.ts` deployment script contains the function `deployLinearAuctionPriceCurve()` that deploys the `LinearAuctionPriceCurve` contract using two parameters. The second parameter is hard-coded in the function instead of using a constant in the `network-constants.ts` script. Replace the use of the hard-coded value with a properly named constant in `network-constants.ts`.

C. Inline Assembly Usage

The use of inline assembly to handle non-conforming `ERC20` token return values raises several concerns:

1. The implementation is tied to low-level details of `solc`'s call structure and token return data. Currently, `solc` does not check that `returndatasize` is `0` in all cases, which may change in a future version of Solidity.
2. There may be Solidity version-related compilation issues with this assembly in the future.

Trail of Bits recommends investigating a separate interface per non-conformant token. This will allow the safe handling of non-conformant tokens, without reliance on low-level details. Furthermore, this will enable the safe handling of other non-standard tokens in the future, which will almost certainly require new interfaces. Trail of Bits recommends a solution similar to the following:

```
pragma solidity 0.4.25;

interface ERC20IncorrectReturnDataSize {
  function transfer(address to, uint value) external;
}

interface ERC20 {
  function transfer(address to, uint value) external returns(bool);
}

contract Contract {

    // set by the owner
    mapping (address => bool) incorrectERC20ReturnDataSize;

    function safeTransfer(address token, address to, uint value) internal {
        if(incorrectERC20ReturnDataSize[token]) {
            ERC20IncorrectReturnDataSize(token).transfer(to, value);
        }
        else {
            require(ERC20(token).transfer(to, value));
        }
    }
}
```

This will allow Set Protocol to handle both standards- and non-standards-compliant tokens in the same contract, without resorting to low-level code that may not be future-proof.

D. ERC20 property-based testing using Echidna

Trail of Bits used Echidna, our property-based testing framework, to find logic errors in the Solidity components of Set Protocol.

Trail of Bits developed custom Echidna testing harnesses for Set Protocol's `SetToken` ERC20 token. These harnesses initialize the tokens and mint an appropriate amount of tokens for two users. It then executes a random sequence of API calls from a `SetToken` contract in an attempt to cause anomalous behavior.

These harnesses includes tests of `ERC20` invariants (*e.g.*, token burn, `balanceOf` correctness, *&c.*), and `ERC20` edge cases (*e.g.*, transferring tokens to oneself and transferring zero tokens). Upon completion of the engagement, these harnesses and their related tests will be delivered to the Set Protocol team.

Figure 1 shows the Solidity source code used to define initialize and test the `SetToken` contract. The script defines a simple token contract used as the single component of the SetToken contract to test. An example of how to run this test with Echidna is show in Figure *2.*

```
import { SetToken } from "contracts/core/tokens/SetToken.sol";
import { ERC20 } from
"openzeppelin-solidity/contracts/token/ERC20/ERC20.sol";
import { ERC20Detailed } from
"openzeppelin-solidity/contracts/token/ERC20/ERC20Detailed.sol";

/**
 * @title SimpleToken
 * @dev Very simple ERC20 Token example, where all tokens are pre-assigned
to the creator.
 * Note they can later distribute these tokens as they wish using
`transfer` and other
 * `ERC20` functions.
 */
contract SimpleToken is ERC20, ERC20Detailed {
    uint256 public constant INITIAL_SUPPLY = 10000 * (10 **
uint256(decimals()));

    /**
     * @dev Constructor that gives msg.sender all of existing tokens.
     */
    constructor () public ERC20Detailed("SimpleToken", "SIM", 18) {
      // Mint the tokens to the owner
```

```
      _mint(msg.sender, INITIAL_SUPPLY);
    }
}

contract TEST {
  SimpleToken vanillaToken;
  SetToken setToken;

  address[] components;
  uint256[] units;

  address testerAddr;
  address otherAddr;
  address ownerAddr;

  uint256 initial_totalSupply;

 constructor() {
    testerAddr = 0xd30a286ec6737b8b2a6a7b5fbb5d75b895f62956;
    otherAddr = 0x67518339e369ab3d591d3569ab0a0d83b2ff5198;
    ownerAddr = address(this);
    // Initialization of the vanilla ERC20 token
    // to list as a standard component.
    vanillaToken = new SimpleToken();
    components.push(address(vanillaToken));
    units.push(10);
    // Initialization of the setToken
    setToken = new SetToken(
      0x0,            // address   _factory
      components,     // address[] _components
      units,          // uint256[] _units
      5,              // uint256   _naturalUnit
      "name",         // string    _name
      "sym"           // string    _symbol
    );

    initial_totalSupply = setToken.totalSupply();

    // describe balances for testing
    setToken.transfer(testerAddr, setToken.totalSupply()/2);
    setToken.transfer(otherAddr, setToken.totalSupply()/2);
  }
  // NOTE: All of these assume the `TEST` contract is the msg.sender,
  // so configuration requires attention to this.
  function totalSupply() public view returns (uint) {
    return setToken.totalSupply();
  }
```

```
  ...

  function balanceOf(address tokenOwner) public view returns (uint balance)
{
    return setToken.balanceOf(tokenOwner);

 function echidna_max_balance() returns (bool) {
    return (balanceOf(testerAddr) <= initial_totalSupply/2 &&
balanceOf(otherAddr) >= initial_totalSupply/2);
  }

  ...

}
```

Figure 1: `test/verification/echidna/SetToken.sol`, which initializes the `SetToken` contract test harness.

```
$ echidna-test ./test/verification/echidna/SetToken.sol TEST --config
./test/verification/echidna/SetToken.yaml
"Analyzing contract: ./test/verification/echidna/SetToken.sol:TEST"
Passed property "echidna_max_balance".
...
```

Figure 2: An example run of Echidna with the `SetToken.sol` test harness, including test results.

It is worth to mention that the same approach could be used to test the behavior of `RebalancingToken` ERC20 token. However, given the short time of the engagement and the complexity of the initialization of such contract, it was not possible to complete.

E. Formal verification using Manticore

Trail of Bits used Manticore, our open-source dynamic EVM analysis tool that takes advantage of symbolic execution, to find issues in the Solidity components of Set Protocol. Symbolic execution allows us to explore program behavior in a broader way than classical testing methods, such as fuzzing.

Trail of Bits used Manticore to determine if certain invalid contract states were feasible. When applied to the `LinearAuctionPriceCurve` contract using the script shown in Figure 1. Manticore identified several parameters that were successfully validated using the `validateAuctionPriceParameters` but make `getCurrentPrice` function revert. The TOB-SP-008 finding details the parameters leading to the revert, and the contract properties affected.

```
from os.path import isdir
from manticore.ethereum import ManticoreEVM

m = ManticoreEVM()

workspace ='test/verification/manticore/LinearAuctionPriceCurve'
assert not isdir(workspace), 'Workspace folder already exists'
m = ManticoreEVM(workspace_url=workspace)
m.verbosity(3)

source_code = '''
    ...
    contract LinearAuctionPriceCurve {
      using SafeMath for uint256;
    ...
    function test(
        uint256 auctionStartTime,
        uint256 auctionTimeToPivot,
        uint256 auctionStartPrice,
        uint256 auctionPivotPrice,
        uint256 block_timestamp
    ) {
        validateAuctionPriceParameters(auctionStartTime,
auctionTimeToPivot, auctionStartPrice, auctionPivotPrice);
        getCurrentPrice(auctionStartTime, auctionTimeToPivot,
auctionStartPrice, auctionPivotPrice, block_timestamp);
     }
}
'''
```

```
user_account = m.create_account(balance=1000, name='user_account')
print("[+] Creating a user account", user_account.name_)

DEFAULT_AUCTION_PRICE_DENOMINATOR = 1000

contract_account = m.solidity_create_contract(source_code,
owner=user_account, name='contract_account',
contract_name='LinearAuctionPriceCurve',
args=[DEFAULT_AUCTION_PRICE_DENOMINATOR])

p1 = m.make_symbolic_value(name="p1")
p2 = m.make_symbolic_value(name="p2")
p3 = m.make_symbolic_value(name="p3")
p4 = m.make_symbolic_value(name="p4")
p5 = m.make_symbolic_value(name="p5")

contract_account.test(p1, p2, p3, p4, p5, value=0)

m.finalize()
print(f"[+] Look for results in {m.workspace}")
```

Figure 1: Manticore testing script which symbolically executes the `validateAuctionPriceParameters` *and* `getCurrentPrice` *functions*

It is worth mentioning that the same approach could be used to verify the behavior of `RebalancingHelperLibrary`. We produced a proof-of-concept to check for reverts in the `calculateTokenFlows` function. However, since this is only an internal function that can be called by different modules, so we are not sure that this can triggered by external users. Given the short time of the engagement and the complexity of the use of such contract, it was not possible to complete.

F. Automatic source code analysis using Slither

Trail of Bits used Slither, a Solidity static analysis framework, to assist with source code analysis of the Set Protocol contracts. Slither contains a set of default detectors to identify security concerns within Solidity code, as well as an underlying framework for working with Solidity source code in an automated fashion.

During this audit, Slither's intermediate representation, `SlithIR`, was used to identify exact equality comparisons performed on the `msg.sender` variable in the codebase (Figure 1). These comparisons were mapped to the functions which contained them, then displayed as output to the user (Figure 2). The goal was to help identify potentially erroneous functions and their dependence on the `msg.sender` variable, due to the complex interactions in the Set Protocol contracts.

```
# Iterate each function in this contract
for function in contract.functions + contract.modifiers:
    # Set this to false for each function so we can perform reporting
    # once the function's node irs has finished being processed.
    function_validates_sender = False
    # Set this to empty so we can build all validation expressions
    # for this function.
    validation_expressions = []
    # Iterate each node in this function.
    for node in function.nodes:
        # This is false by default, proven true by iterating the node
        # irs.
        node_validates_sender = False
        for ir in node.irs:
            # The ir must be a binary op that performs an `==` compare
            if isinstance(ir, Binary) and ir.type == BinaryType.EQUAL:
                var_names = [ir.variable_left.name, ir.variable_right.name]
                if "msg.sender" in var_names:
                    # Flag this function as correctly validating sender
                    function_validates_sender = True
                    # Flag this node as validating the sender
                    node_validates_sender = True
        # If the node validates the sender, add it to the list of
        # expressions we are tracking.
        if node_validates_sender:
            validation_expressions.append(node)
```

Figure 1: An excerpt of the `check_sender_validations` *function which identifies all* `msg.sender` *exact equality comparisons*

```
...
Contract: IssuanceOrderModule
        Validated functions:
                - cancelOrder:
                        - EXPRESSION require(bool,string)(_order.makerAddress ==
msg.sender,IssuanceOrderModule.cancelOrder: Unauthorized sender)
Contract: StandardSettleRebalanceLibrary
        Validated functions:
                - None
...
Contract: Vault
        Validated functions:
                - isOwner:
                        - RETURN msg.sender == _owner
```

Figure 2: A snippet of example output of the `slither-set` *script*

G. Fix Log

Trail of Bits performed a retest of the Set Protocol system during March 25-29, 2019. Set Protocol provided fixes and supporting documentation for the findings outlined in their most recent security assessment report. Each finding was re-examined and verified by Trail of Bits.

Set Protocol rearchitected and centralized their data validation logic, resulting in much more comprehensive and thorough validation. They also rewrote their deployment scripts to run correctly with no manual intervention. A detailed log of their responses to discovered issues follows below.

Fix Log Summary

#	Title	Severity	Status
1	Inline assembly is used to validate external contract calls	Medium	Not fixed
2	SetToken can reference itself as a component	Informational	Not fixed
3	SetToken components have limited upgradability	Medium	Not fixed
4	TimeLockUpgrade's timeLockPeriod remains default post-deployment	High	Fixed
5	Race condition in the ERC20 approve function may lead to token theft	High	Not fixed
6	Deployments and migrations require further testing	High	Fixed
7	Whitelist validations are not consistently used	Medium	Fixed
8	Inadequate data validation in price libraries could result in unexpected reverts	Medium	Fixed
9	0x exchange wrapper is unable to increase approval for relay fees	Medium	Fixed

10	Current governance structure introduces counterparty risk	Informational	Not fixed
11	Component rebalance effectively pauses parent issuance	Medium	Not fixed
12	Solidity compiler optimizations can be dangerous	Undetermined	Not fixed
13	Insufficient validation of the rebalanceInterval parameter could produce a revert in the propose function	Medium	Fixed
14	The ether quantity in the LogPayableExchangeRedeem event cannot be trusted	Undetermined	Fixed
15	Insufficient input validation in ExchangeIssuanceModule functions	Medium	Fixed
16	hasDuplicate runs out of gas when the input list is empty	Medium	Fixed
17	executeExchangeOrders fails to properly validate repeated exchanges	High	Fixed

Detailed Fix Log

This section includes brief descriptions of issues that were fully addressed.

Finding 4: TimeLockUpgrade's timeLockPeriod remains default post-deployment
This functionality is documented in the whitepaper, as is the planned migration path away from it.

Finding 6: Deployments and migrations require further testing
Set Protocol rewrote the scripts, which now work without manual intervention.

Finding 7: Whitelist validations are not consistently used
Set Protocol rearchitected their validation logic to be more centrally located.

Finding 8: Inadequate data validation in price libraries could result in unexpected reverts
Set Protocol has reengineered their validation logic, and the testcase given is currently infeasible, as RebalancingSetToken start times are hardcoded to zero.

Finding 9: 0x exchange wrapper cannot increase approval for relay fees
As it happens, if the allowance is the maximum UINT, the ZRX token implementation does not decrement balances during transfer.

Finding 13: Insufficient validation of the rebalanceInterval parameter could produce a revert in the propose function
This issue only allows users to spend their own gas creating and manipulating pathological sets, it does not pose a risk to the protocol. The use of SafeMath removes issues that could occur due to overflow.

Finding 14: The ether quantity in the LogPayableExchangeRedeem event cannot be trusted
LogPayableExchangeRedeem now logs the rebalancing set quantity, not Ether, mitigating this finding.

Finding 15: Insufficient input validation in ExchangeIssuanceModule functions
Set Protocol has refactored and improved their validation logic, fixing this issue.

Finding 16: hasDuplicate runs out of gas when the input list is empty
Set Protocol added logic to catch this corner case.

Finding 17: executeExchangeOrders fails to properly validate repeated exchanges
Set Protocol has refactored and improved their validation logic, fixing this issue.

Detailed Issue Discussion

Responses from Set Protocol for partial or unfixed issues are included as quotes below.

Finding 1: Inline assembly is used to validate external contract calls

> *Set Protocol intends to migrate to OpenZeppelin's SafeERC20 library, which addresses this finding, as soon as their implementation handles reversions on functions with numeric return values.*

Finding 2: SetToken can reference itself as a component

> *Set Protocol is not concerned about these cyclic tokens, since issuing a set requires providing all underlying components, making issuing cyclic sets impossible.*

Finding 3: SetToken components have limited upgradability

> *Set Protocol's newest whitepaper contains plans to add a TokenMigrationModule to address this when necessary.*

Finding 5: Race condition in the ERC20 approve function may lead to token theft

> *Set Protocol's newest whitepaper contains a discussion on the ERC20 approve race condition issue, warning users to utilize increaseApproval and decreaseApproval when possible. Set Protocol has taken this into account and accepted the risk.*

Finding 10: Current governance structure introduces counterparty risk

> *Set Protocol accepts the risk, and is creating a dashboard so that this information is publicly accessible.*

Finding 11: Component rebalance effectively pauses parent issuance

> *Only whitelisted tokens can be used as components of rebalancing sets. Set Protocol is taking care when whitelisting tokens to avoid allowing the creation of pathological sets.*

Finding 12: Solidity compiler optimizations can be dangerous

> *Set Protocol acknowledges the risk, but currently their contract is too complex to be deployable without optimizations.*