Smart Contract Source Code Audit Sovryn

Prepared for Sovryn • December 2020

v201218

1. Table Of Contents

1. Table Of Contents

2. Executive Summary

3. Introduction

4. Assessment

Phase 1: Review of all changes introduced to upstream projects bZx and Bancor

Phase 3: General review of the lending, trading and borrowing processes

5. Conclusions and Recommendations

6. Summary of Findings

7. Remediations

8. Findings

SVN-001 - Price feed oracle fake timestamp

SVN-002 - WRBTC ERC20 approve front running

SVN-003 - internalSwap function name is misleading

SVN-004 - Infinite transfer allowance

SVN-010 - Missing or numeric non descriptive error messages

SVN-011 - Transaction size and slippage limits not enforced for external swaps

SVN-012 - Leftover code from debugging

SVN-013 - Function _totalDeposit doesn’t revert when the precision is 0

9. Disclaimer

2. Executive Summary

In October 2020, Sovryn engaged Coinspect to perform a source code review of their new decentralized Bitcoin trading and lending platform. The objective of the audit was to evaluate the security of their smart contracts.

The code reviewed was found to be clear, well written, and properly documented. The modifications performed to the forked projects did not introduce any vulnerabilities. However, Coinspect observed the oracle integration implementation weakens the system security and could be abused by attackers to manipulate the price feeds.

Moreover, the protocol is dependent on third party oracle providers, whose security should be evaluated and taken into consideration when deciding to use the Sorvyn platform. The oracles are trusted by Sovryn and are a single point of failure for the whole system.

The following issues were identified during the assessment:

High Risk	Medium Risk	Low Risk	Informational
1	**0**	**4**	**3**

During December 2020 Coinspect verified the fixes developed by the Sovryn team were correct. Detailed information regarding these fixes and the current status for each finding can be found in 7. Remediations.

3. Introduction

Sovryn's goal is to enable lending, borrowing and margin trading in the RSK blockchain.

The project architecture is composed of the following components:

1. **Core protocol:** The Core protocol is a bZx - A Protocol For Tokenized Margin Trading and Lending protocol fork. The most important change introduced by Sovryn is the switch to using their **oracle-based AMM** instead of Kyber. Also, some protocol parameters were modified, such as: rollover rewards and minimum utilization rate on interest calculation. This component can be found in https://github.com/DistributedCollective/Sovryn-smart-contracts.
2. **Oracle-based Automated Market Maker:** This component is a Bancor Network liquidity protocol fork. The most important modification introduced by Sovryn is the switch from Chainlink oracles to the Money on Chain ones. This component is located in https://github.com/DistributedCollective/oracle-based-amm
3. **Watcher:** this off-chain component is responsible for the liquidation and rollover of open positions. It reads all open positions from the Sovryn smart contracts and continuously monitors for changes, then triggers transaction submissions when appropriate. This component can be found in https://github.com/DistributedCollective/Sovryn-Watcher/tree/audit-coinspect

The whole engagement was structured in phases:

1. **Phase 1**: review of all changes introduced to upstream projects bZx and Bancor.
2. **Phase 2**: review of the off-chain Watcher component.
3. **Phase 3**: review of the lending, borrowing and trading flows.

This report documents phases 1 and 3 of the audit.

The audit started on October 27th and was conducted on the following Git repositories:

1. https://github.com/DistributedCollective/Sovryn-smart-contracts as of commit `86008054558bd7ce02e6b3b0547c681b62ecd4fc` of **October 26th.**
2. https://github.com/DistributedCollective/oracle-based-amm as of commit `8b6504406b89ad24bf4e0f5ff97037bf798b59c8` of **October 9th.**

The scope of the audit's **Phase 1** was limited to the following pull requests as requested by Sovryn:

- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/12
- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/13
- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/24
- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/28
- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/30
- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/31
- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/34
- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/35
- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/42
- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/44
- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/52
- https://github.com/DistributedCollective/oracle-based-amm/pull/1
- https://github.com/DistributedCollective/oracle-based-amm/pull/4
- https://github.com/DistributedCollective/oracle-based-amm/pull/8
- https://github.com/DistributedCollective/oracle-based-amm/pull/10

Neither the upstream projects' security nor the Money on Chain oracle infrastructure were evaluated during this audit.

4. Assessment

Phase 1: Review of all changes introduced to upstream projects bZx and Bancor

The following list including all modifications introduced by Sovryn to the forked repositories was provided by the team and were reviewed during this phase of the engagement:

a. **Sovryn Protocol**:
   1. Replaced the Kyber connector with a connector to Sovryn Swap
   2. Use MoC as oracle for the price feed instead of Chainlink / Kyber
   3. Disabled flash loans
   4. LoanId creation refactoring
   5. Removal of hard coded addresses
   6. Changed the rollover reward
   7. Lowered the minimum utilization rate on interest calculation
   8. Introduced a multisig owner

b. **Sovryn Swap**:
   9. Split up the factory function which deploys the liquidity pool v2 converter
   10. Use MoC as oracle instead of Chainlink

Additionally, the new wrapped `RBTC` and the `RBTCWrapperProxy` contracts were added during the audit and were reviewed as per the client's request.

The following sections explore each of these code modifications, and provide a brief description and audit notes for each of them.

1. Replaced the Kyber connector with a connector to Sovryn Swap

This is the project's biggest set of changes.

Sovryn created a new connector contract, which connects to Sovryn's oracle-based AMM and replaces the existing Kyber swap connector.

The new `SwapsImplSovrynSwap` contract implements the `ISwapsImpl` interface and is responsible for token swapping, these are its most relevant characteristics:

1. Uses OpenZeppelin's `SafeERC20` for token transfers.
2. The source token estimation was improved to account for rounding in the AMM.
3. Source code documentation was improved.
4. It keeps a reference to the Sovryn Swap Network contract, only the connector contract owner is allowed to modify it.
5. Relies on the Sovryn Swap Network to perform token conversions and calculate exchange rates.
6. Bubbling of errors from Sovryn Swap network is allowed (this differs from the Kyber connector implementation which does not allow it)
7. If the `returnToSenderAddress` parameter is not the protocol itself, any source token remaining after the swap are sent back to this address

Because the oracle-based AMM does not have the option to pass a maximum amount of destination tokens, the rollover function in `LoanClosings` was adapted. After the interest was swapped, the excess gets swapped back in the new function `_swapBackExcess`. Sovryn optimized the `LoanClosings` contract to swap back excess (from the rollover and the borrower scenarios) only if the amount is big enough to justify the swap transaction, usually the excess is a fraction of a cent and not worth the extra gas cost. The hard coded 0.00001 RBTC is used as the threshold value for borrower excess. The new function `worthTheTransfer` is responsible for obtaining the exchange rate from the priceFeeds contract and comparing the resulting value with the threshold. In the `_coverPrincipalWithSwap` scenario, when the excess is under the threshold limit, it is always sent back to the lender. But in the `_rollover` scenario, excess under threshold is kept as a protocol lending fee.

Coinspect reviewed the following pull requests:

- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/30
- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/52

2. Use MoC as oracle for the price feed instead of Chainlink / Kyber

The following modifications were introduced:

1. Added a price feed contract `PriceFeedsMoC` which connects to the MoC oracle: https://github.com/money-on-chain/Amphiraos-Oracle/blob/master/contracts/medianizer/medianizer.sol, replacing Kyber and Chainlink as price feed sources.
2. The value retrieved from the price feed contract `latestAnswer` function is `uint256` instead of `int256`.
3. Added a base token parameter to the price feed constructor.
4. Removed gas price retrieval function `getFastGasPrice`.

Coinspect observed that in the current implementation, the feed contract lacks the ability to know when was the last time the oracle was updated because the MoC oracle does not provide that information back to the consumer contract. This issue is fully described in Price feed oracle fake timestamp.

Coinspect verified only the `PriceFeedMoC` contract owner can set the MoC oracle contract address.

Coinspect reviewed the following pull requests:

1. https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/28
2. https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/42

3. Disabled flash loans

Sovryn commented out the function `flashBorrow` in order to disable the flash loan functionality for the MVP release. Also, the `reentrancyGuard` modifier was re-enabled for the `marginTrade` and `borrow` functions. This modifier will have to be removed again once the flash loans are enabled.

It is worth noting that even if flash loans are disabled in the Sovryn platform, they could still be offered by another platform enabling attackers to utilize them in order to exploit Sovryn.

This change was introduced in the following pull request:

- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/13

4. loanId creation refactoring

Sovryn modified the way the `loanId` is calculated when a new loan is opened. A per user nonce is used instead of the block timestamp, in addition to the lender, borrower and `loanParamsLocal.id.`
Coinspect reviewed this change and concluded that making the `loanId` deterministic does not represent a risk to the platform's security.

This change is introduced in the following pull request:

- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/24

5. Removal of hardcoded addresses

Sovryn moved WETH and the protocol token addresses from `Constants.sol` to `State.sol`; setters were added to the price feed contracts. The `LoanToken` constructor is now passed the `sovrynContractAddress` and `wbtcTokenAddress` parameters which were previously hard coded.

Coinspect verified that only the contract owner is able to access the new configuration setters.

This change is implemented in the following pull request:

- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/12

6. Changed the rollover reward

The `GasTokenUser` contract was removed together with all the existing gas rebate logic. The loan rollover reward which was before based on the transaction gas cost is now replaced with the following calculation instead:

```
uint256 public rolloverBaseReward = 16800000000000;
// Rollover transaction costs around 0.0000168 rBTC, it is denominated in wRBTC
uint256 public rolloverFlexFeePercent = 0.1 ether;                      // 0.1%

return rolloverBaseRewardInCollateralToken.mul(2) // baseFee
      .add(positionSizeInCollateralToken.mul(rolloverFlexFeePercent).div(10 ** 20));
     // flexFee = 0.1% of position size
```

Note the new `RewardHelper` contract relies on the `priceFeeds` oracle contract in order to calculate the reward in the corresponding collateral token.

This change is introduced by the following pull request:

- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/31

7. Lowered the minimum utilization rate on interest calculation

Previously, a minimum utilization rate of 80% was hardcoded and could not be adjusted as needed, now it can be parametrized together with `targetLevel`, `kinkLevel` and the `maxScaleRate` parameters using the new function `setDemandCurve`. The interest calculation logic in function `_nextBorrowInterestRate2` was updated to use the parameterized values instead of hardcoded ones. This change was merged from the bZx repository.

This change is introduced by the following pull requests:

- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/34

- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/44

8. Introduced a multisig owner

In order to replace the single address that owned all the Sovryn protocol contracts, a 2of3 multisig wallet was introduced. This multisig is intended to be used while the governance model is being developed. The deployment script was modified to transfer ownership of the Sovryn protocol contract to the multisig.

This change was introduced by the following pull request:

- https://github.com/DistributedCollective/Sovryn-smart-contracts/pull/35

9. Split up the factory function which deploys the liquidity pool v2 converter

The function `newConverter` in the `ConverterRegistry` contract needed to be split in order to be able to deploy it on RSK because of the network's gas limit of 6.8M. A new function, `setupConverter` was added to complete the deployment.

PR8 correctly fixed an issue found by a previous security audit performed by a different team by checking the user finishing the contract deployment is the same that initiated it.

This change was introduced by the following pull requests:

- https://github.com/DistributedCollective/oracle-based-amm/pull/1
- https://github.com/DistributedCollective/oracle-based-amm/pull/8

10. Use MoC as oracle instead of Chainlink

Sovryn modified the AMM to utilize the Money on Chain oracle infrastructure running in the RSK network instead of Chainlink as the source for off-chain price feeds.

Even though the change is straightforward implementation-wise, Coinspect observed the `latestTimestamp` function does not return a value obtained from the oracle as expected, but the block timestamp. This behavior prevents the oracle consumer from using this information in order to make a decision regarding the validity of the off-chain data and is fully documented in Price feed oracle fake timestamp.

Additionally, Coinspect verified only the contract owner can set the MoC oracle contract address.

This change was introduced by the following pull request:

- https://github.com/DistributedCollective/oracle-based-amm/pull/4

11. Wrapped RBTC and RBTCWrapperProxy

Sovryn added the new `RBTCWrapperProxy` and `WRBTC` contracts. The wrapped RBTC contract gives depositors one WRBTC token per each RBTC sent to it. In the same way, it allows withdrawing one RBTC for each WRBTC token burned.

The `RBTCWrapperProxy` enables users to:

1. Add liquidity to the pools reserves in exchange for pool tokens
2. Remove liquidity

3. Convert their tokens

The `RBTCWrapperProxy` receives RBTC from the user and wraps it WRBTC tokens before depositing it into the liquidity pool; then it sends the pool tokens back to the user.

The `convertByPath` function can:

1. Receive RBTC as value, which gets wrapped and swapped
2. Convert any token to WRBTC

This change was introduced by the following pull request:

- https://github.com/DistributedCollective/oracle-based-amm/pull/10

Phase 3: General review of the lending, trading and borrowing processes

During this phase of the engagement, Coinspect reviewed how all Sovryn components integrate, reviewed the deployment procedures, and tested user-Sovryn interactions.
This process focused on the following contracts and entry points as requested by the Sovryn team:

1. `LoanTokenLogicStandard.sol`
    a. `marginTrade`
    b. `borrow`
    c. `mint`
    d. `burn`
2. `LoanTokenLogicWrbtc.sol:`
    a. `mintWithBTC`
    b. `burnToBTC`
3. `LoanClosing.sol`
    a. `closeWithSwap`
    b. `closeWithDeposit`
    c. `liquidate`
    d. `rollover`

While reviewing the deployment and setup process, Coinspect auditors observed the contract owning the protocol has unlimited powers including:

1. Upgrading all Sovryn protocol smart contracts.
2. Withdrawing all funds.
3. Pausing/unpausing the protocol.
4. Changing all protocol parameters (interest curve, AMM connector, oracles, etc).

Coinspect recommends splitting administrative roles in order to minimize damage in case one role is compromised.

The tests included in the https://github.com/DistributedCollective/Sovryn-smart-contracts repository were reviewed. These tests have been developed using the `brownie` framework, and are currently being migrated to the `truffle` framework. Besides some sporadic errors related to the `brownie` framework, all tests included pass. The tests are intended to verify the basic functionality of the protocol is correct, and no complex scenarios are included. For example: all tests consist of one lender and one borrower.
Coinspect auditors were unable to obtain a valid coverage report, when coverage is enabled most tests fail. The Sovryn team is aware of this fact and this is one of the reasons why the brownie framework is being abandoned. It is important that the ability to evaluate the tests coverage is recovered in order to obtain a clear view of what execution paths are being exercised and which tests need to be improved or created.

Coinspect auditors found not all the platform's entry points enforce the same security limits for maximum transaction amount and slippage. This is detailed in Transaction size and slippage limits not enforced for external swaps.

Regarding slippage, there is a hardcoded slippage limit of 5%, enforced by the function `checkPriceDisagreement` in the `PriceFeeds` contract, for all borrowing, lending and margin trading originated swaps performed in the Sovryn exchange:

```
        uint256 public maxDisagreement = 5 * 10**18;
        // % disagreement between swap rate and reference rate
```

This means all operations in the Sovryn exchange are subject to losing up to 5% from the internal swap performed.

It is worth noting no attempt at limiting swaps was observed in the oracle-based AMM project either. However, the size of the pools can be restricted by the contract owner, and this maximum liquidity pool size is enforced by the `addLiquidity` function. This limit is not hardcoded though, and depends on the deployment and configuration actions performed by the contract owner for each pool. *By default, new pools are created with unlimited staked balance.*

5. Conclusions and Recommendations

In respect to the smart contracts reviewed, the changes introduced to the forked projects did not introduce any security vulnerabilities and were well documented. The oracle integration did weaken the platform overall security by voiding the last update timestamp checks that were in place.

The following list sums up the most important recommendations from this audit:

1. Continue Improving the oracle integration as this could be seen as the weakest link in the platform.
2. Add tests for oracle's worst case scenarios.
3. Improve end to end testing to include complex scenarios (e.g., chain reorganizations, network congestion, multiple lenders and borrowers, trade operations with sizes around maximum limits, slippage).
4. Fix testing coverage reporting.
5. Create administrative roles with different sets of privileges.
6. Clearly document the upgradable nature of the protocol and the operations accessible by the contract's owners.
7. Constantly monitor vulnerabilities reported in the upstream projects and backport the changes as needed.

6. Summary of Findings

ID	Description	Risk	Fixed
SVN-001	Price feed oracle fake timestamp	High	✔
SVN-002	WRBTC ERC20 approve front running	Low	✘
SVN-003	internalSwap function name is misleading	Info	✘
SVN-004	Infinite transfer allowance	Low	✘
SVN-010	Missing or numeric non descriptive error messages	Info	✘
SVN-011	Transaction size and slippage limits not enforced for external swaps	Low	✘
SVN-012	Leftover code from debugging	Info	✔
SVN-013	Function _totalDeposit doesn’t revert when the precision is 0	Low	✔

7. Remediations

During December 2020 Coinspect verified the findings that Sovryn decided to address had been correctly fixed.

The following table lists the findings that were fixed and the corresponding pull requests:

ID	Description	Pull Request
SVN-001	Price feed oracle fake timestamp	PR #76 PR #15
SVN-012	Leftover code from debugging	PR #84
SVN-013	Function _totalDeposit doesn't revert when the precision is 0	PR #85

SVN-001 has been mitigated by adding a new price source, an oracle contract provided by the RSK team, for the WRBTC price. This oracle in the Sovryn Protocol will be used to check price divergence between this new feed and the exchange rate obtained from the AMM component. This change will not affect transactions performed directly in the AMM. Additionally, code in the AMM repository was modified to use the latest publication block number (that will be provided by the MoC oracle in the future) to calculate the latest oracle update timestamp. This fix is not currently deployed and has only been tested using a mock contract. Coinspect has not reviewed the recently introduced RSK oracle infrastructure.

Regarding SVN-011, the vulnerable contract has not been deployed so Sovryn is currently not exposed to any risk related to this finding. The Sovryn team will implement a price divergente check in the contract before it is deployed. The limit check is considered unnecessary for this contract as funds are never stored in it.

The Sovryn team decided not to fix SVN-002 and is considering if SVN-003 and SVN-004 will be fixed; these are all low risk findings.

8. Findings

SVN-001 Price feed oracle fake timestamp

Total Risk	Impact	Location
High	High	MocBTCToUSDOracle.sol
	Likelihood	
Fixed ✔	High	

Description

Sovryn utilizes the MoC oracles platform for its price feeds.

First, Coinspect auditors observed that, in the oracle-based AMM component, the function `latestTimestamp` in the `MoCBTCToUSDOracle` contract fakes the timestamp of the latest price update to the block timestamp:

```
/**
  * @dev returns the USD/BTC update time.
  *
  * @return always returns current block's timestamp
*/
function latestTimestamp() external view returns (uint256) {
    return now; // MoC oracle doesn't return update timestamp
}
```

As the code comment suggests, the current version of the MoC oracle medianizer contract does not provide the last update timestamp.

This results in the `lastUpdateTime` and `lastRateAndUpdateTime` functions in `PriceOracle.sol` (which are used internally by the AMM) being useless as well:

```
   /**
       * @dev returns the timestamp of the last price update the rates are returned as
numerator (token1) and denominator
     * (token2) for accuracy
     *
     * @return timestamp
   */
   function lastUpdateTime()
       public
       view
       returns (uint256) {
       // returns the oldest timestamp between the two
       uint256 timestampA = tokenAOracle.latestTimestamp();
       uint256 timestampB = tokenBOracle.latestTimestamp();

       return  timestampA < timestampB ? timestampA : timestampB;
   }
```

Because of this, the AMM will always depend on the rate returned by the MoC oracle for weight rebalancing purposes. The original AMM behavior falls back to the AMM internal rate if the price has not been updated recently.

Secondly, the auditors observed, on the Sovryn Core Protocol component side, the oracle consumer contract `PriceFeedsMoC.sol` retrieves the latest answer value and the `hasValue` boolean flag (which allows the oracle to signal the consumer that the provided value is not considered valid) from the MoC oracle `Medianizer` contract:

```
 function latestAnswer()
     external
     view
     returns (uint256)
 {
     (bytes32 value, bool hasValue) = Medianizer(mocOracleAddress).peek();
     require(hasValue, "Doesn't have a value");
     return uint256(value);
 }
```

This is how `_queryRate` is implemented in `PriceFeeds.sol`:

```
function _queryRate(
     address sourceToken,
     address destToken)
     internal
     view
     returns (uint256 rate, uint256 precision)
 {
     require(!globalPricingPaused, "pricing is paused");

     if (sourceToken != destToken) {
         uint256 sourceRate;
         if (sourceToken != address(baseToken) && sourceToken != protocolTokenAddress) {
             IPriceFeedsExt _sourceFeed = pricesFeeds[sourceToken];
             require(address(_sourceFeed) != address(0), "unsupported src feed");
             sourceRate = _sourceFeed.latestAnswer();
             require(sourceRate != 0 && (sourceRate >> 128) == 0, "price error");
```

As a consequence of the above observations, if the MoC oracle fails to be updated during a period of time (and the `hasValue` flag is not set to false), Sovryn components will continue operating with a potentially outdated exchange rate, which would result in liquidity being drained from the AMM pools for example. The Sovryn AMM thus deposits all trust in the oracle implementation ability to decide if the information provided is valid or not.

A few potential reasons for MoC oracles to end up with outdated values are:

- Network congestion scenario (natural or attacker induced) where the update transactions fail to get mined.
- Oracle update agents running out of gas
- Oracle updates not fast enough to cope with rapid changes in price variations,
- System administration issues.

A full analysis of the MoC oracle implementation and infrastructure was beyond the scope of this audit.

Recommendations

Coinspect recommends the Sovryn platform improves their integration of off-chain oracles:

1. Request MoC to add last update timestamp information to their oracles. Use this information to decide if the obtained value should be trusted or not.
2. Consider adding redundancy for price feeds
3. Research latest advances in blockchain oracle technology (such as the new Maker Oracle Security Modules)

SVN-002 WRBTC ERC20 approve front running

Total Risk	Impact	Location
Low	Medium	rbtcwrapperproxy/WRBTC.sol
Fixed	Likelihood	
✘	Low	

Description

The wrapped RBTC token contract suffers from a well known ERC20 standard security vulnerability that takes place when the token transfer allowance is modified: an attacker can front run the approve transaction to transfer the original allowed amount of tokens (N) before the allowance is changed, and then, after the approve transaction takes place, the attacker can again transfer more tokens (M), obtaining as a result more tokens than the toker owner intended (N+M instead of M) [1].

Recommendation

Add the functions `increaseApproval` and `decreaseApproval` to the WRBTC contract, using as a template the implementations in the OpenZeppelin library [2].

References

[1] https://github.com/ethereum/EIPs/issues/20#issue comment-263524729
[2] https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/token/StandardToken.sol#L70

SVN-003 internalSwap function name is misleading

Total Risk **Info**	Impact None	Location swaps/connectors/SwapsImplSovrynSwap.sol
Fixed ✘	Likelihood None	

Description

The only state changing function in the `SwapsImplSovrynSwap` contract is `internalSwap`:

```
function internalSwap(
       address sourceTokenAddress,
       address destTokenAddress,
       address receiverAddress,
       address returnToSenderAddress,
       uint256 minSourceTokenAmount,
       uint256 maxSourceTokenAmount,
       uint256 requiredDestTokenAmount)
       public
       returns (uint256 destTokenAmountReceived, uint256 sourceTokenAmountUsed)
```

Using *internal* in a public function name is confusing, and could result in a developer incorrectly assuming the function can not be accessed from outside the contract, leading to security vulnerability.

The same happens with the `internalExpectedRate` function, though this function does not modify state.

```
   function internalExpectedRate(
       address sourceTokenAddress,
       address destTokenAddress,
       uint256 sourceTokenAmount)
       public
       view
       returns (uint256)
   {
```

Recommendation

Even though this issue does not represent a security risk right now, Coinspect recommends modifying the functions name to improve code readability and prevent future mistakes.

SVN-004 Infinite transfer allowance

Total Risk	Impact	Location
Low	Medium	swaps/connectors/SwapsImplSovrynSwap.sol
Fixed	Likelihood	
✘	Low	

Description

The `SwapsImplSovrynSwap` contract allows the oracle-based AMM component to transfer unlimited amounts of its tokens. This allowance is never revoked. This behaviour was inherited from the original Kyber connector.

The function `allowTransfer` is called by `internalSwap` everytime a token swap is performed:

```
   /**
    * check is the existing allowance suffices to transfer the needed amount of tokens.
    * if not, allows the transfer of an arbitrary amount of tokens.
    * @param tokenAmount the amount to transfer
    * @param tokenAddress the address of the token to transfer
    * @param sovrynSwapNetwork the address of the sovrynSwap network contract.
    * */
   function allowTransfer(
       uint256 tokenAmount,
       address tokenAddress,
       address sovrynSwapNetwork)
       internal
   {
       uint256 tempAllowance = IERC20(tokenAddress).allowance(address(this),
sovrynSwapNetwork);
       if (tempAllowance < tokenAmount) {
           IERC20(tokenAddress).safeApprove(
               sovrynSwapNetwork,
               uint256(-1)
           );
       }
   }
```

An infinite allowance implies an implicit trust in the oracle-based AMM component, which is not necessary, and has potential for abuse.

Recommendation

Coinspect suggests approving only the amount required for the current swap in order to contain the impact of a potential vulnerability in the oracle-based AMM component.

SVN-010 Missing or numeric non descriptive error messages

Total Risk	Impact	Location
Info	None	connectors/loantoken/*.sol
Fixed	Likelihood	
✘	None	

Description

On several occasions, the error messages returned to users are missing, or they are numeric and not self explanatory. This can be seen in `require` and `_safeTransfer` calls, these a few examples in `LoanTokenLogicStandard.sol`:

```
require(_loanTokenAddress != collateralTokenAddress, "26");

require (sentAmounts[1] != 0, "25");

_safeTransfer(_loanTokenAddress, receiver, withdrawalAmount, "");

_safeTransferFrom(collateralTokenAddress, msg.sender, sovrynContractAddress,
collateralTokenSent, "28-b");
```

These errors strings are hard to understand from the user point of view and/or while reading the source code.

Recommendations

Replace the numeric error messages with easier to understand string constants. Also, it is important to include the reason string in tests that verify a revert, in order to make sure that the transaction is reverted by the expected reason and not because of some other problem.

SVN-011 Transaction size and slippage limits not enforced for external swaps

Total Risk
Low

Fixed
✘

Impact
Medium

Likelihood
Low

Location
SwapsExternal.sol
SwapsImplSovrynSwap.sol

Description

While operations performed though the Sovryn exchange are limited in size and protected from arbitrary slippage conditions, the user accessible `SwapsExternal` contract permits unbounded swaps with no slippage checks enforced.

The `swapExternal` public function in the `SwapsExternal` smart contract can be invoked by anybody (without need for having an open position), as it name indicates, to swap tokens. This function relies on the internal function `_swapsCall` located in the `SwapsUser` contract, which is the function used by the `_loanSwap` function in the same contract.

`_loanSwap` is the function used by all the trading/lending token logic (e.g., `LoanClosings.sol`, `LoanMaintenance.sol and LoanOpenings.sol`) and enforces limits for:

1. Maximum amount of source token swapped: 50 RBTC as defined in `State.sol`:

```
uint256 public maxSwapSize = 50 ether;
// maximum support swap size in BTC
```

2. Maximum price slippage: 5% as defined in `State.sol`:

```
uint256 public maxDisagreement = 5 * 10**18;
// % disagreement between swap rate and reference rate
```

This is the relevant code in `_loanSwap`:

```
    function _loanSwap(
        bytes32 loanId,
        address sourceToken,
        address destToken,
        address user,
        uint256 minSourceTokenAmount,
        uint256 maxSourceTokenAmount,
        uint256 requiredDestTokenAmount,
        bool bypassFee,
        bytes memory loanDataBytes)
        internal
        returns (uint256 destTokenAmountReceived, uint256 sourceTokenAmountUsed, uint256 
sourceToDestSwapRate)
    {
        (destTokenAmountReceived, sourceTokenAmountUsed) = _swapsCall(
            [
                sourceToken,
                destToken,
                address(this), // receiver
                address(this), // returnToSender
                user
            ],
            [
                minSourceTokenAmount,
                maxSourceTokenAmount,
                requiredDestTokenAmount
            ],
            loanId,
            bypassFee,
            loanDataBytes
        );

        // will revert if swap size too large
        _checkSwapSize(sourceToken, sourceTokenAmountUsed);

        // will revert if disagreement found
        sourceToDestSwapRate = IPriceFeeds(priceFeeds).checkPriceDisagreement(
            sourceToken,
            destToken,
            sourceTokenAmountUsed,
            destTokenAmountReceived,
            maxDisagreement
        );
```

However, the `swapExternal` function implementation does not perform any of those checks before calling the AMM contract (via a `swapImpl.delegatecall` to the `internalSwap` function in the AMM connector located in the `SwapsImplSovrynSwap` contract) for the swap in the same `_swapsCall` used above:

```
function swapExternal(
    address sourceToken,
    address destToken,
    address receiver,
    address returnToSender,
    uint256 sourceTokenAmount,
    uint256 requiredDestTokenAmount,
    bytes calldata swapData)
    external
    payable
    nonReentrant
    returns (uint256 destTokenAmountReceived, uint256 sourceTokenAmountUsed)
{
    require(sourceTokenAmount != 0, "sourceTokenAmount == 0");

    if (msg.value != 0) {
        if (sourceToken == address(0)) {
            sourceToken = address(wrbtcToken);
        }
        require(sourceToken == address(wrbtcToken), "sourceToken mismatch");
        require(msg.value == sourceTokenAmount, "sourceTokenAmount mismatch");
        wrbtcToken.deposit.value(sourceTokenAmount)();
    } else {
        IERC20(sourceToken).safeTransferFrom(
            msg.sender,
            address(this),
            sourceTokenAmount
        );
    }

    (destTokenAmountReceived, sourceTokenAmountUsed) = _swapsCall(
        [
            sourceToken,
            destToken,
            receiver,
            returnToSender,
            msg.sender // user
        ],
        [
            sourceTokenAmount, // minSourceTokenAmount
            sourceTokenAmount, // maxSourceTokenAmount
            requiredDestTokenAmount
```

```
                ],
                0, // loanId (not tied to a specific loan)
                false, // bypassFee
                swapData
            );
```

Coinspect found the public function `internalSwap` function in the AMM connector located in the `SwapsImplSovrynSwap` contract also allows bypassing the trade/borrow related swaps security limits.

As a result, *the protection mechanisms put in place to limit transaction sizes and slippage in the Sovryn platform are not consistent among all user accessible interfaces.* If a vulnerability is discovered in the platform, this no-limits public function could be abused to exploit it, bypassing the limits imposed by the other functions.

For a user to be exposed to this vulnerability, he would need to be tricked into using the unprotected function instead of the regular mechanism exposed by the dApp frontend.

Status

The Sovryn team explained that this issue is mitigated by the following reasons:

1. This contract is not currently deployed in mainnet
2. Limits are not considered necessary because the contract does not store funds

However, the price divergence check will be added to the contract before its deployment.

Recommendations

In order to bring consistency to the whole Sovryn platform protection mechanism, Coinspect recommends mirroring the checks in place in the `_loanSwap` function to the `swapExternal` and `internalSwap` functions (or moving all checks to the `internalSwap` function if that is the only entry point to the AMM).

If the external swap functionality needs to remain unlimited for some reason, Coinspect suggests clearly documenting the different limits imposed by each component to improve transparency in that respect.

SVN-012 Leftover code from debugging

Total Risk	Impact	Location
Info	None	connectors/loantoken/LoanTokenLogicStandard.sol
Fixed	Likelihood	
✔	None	

Description

The function `_updateCheckpoints` emits the `Debug` event:

```
emit Debug(
        slot,
        _currentProfit,
        _currentPrice
);
```

Recommendations

This looks like a leftover from a debug session, and it was probably not intended to commit it to the git repository. It is recommended to remove the event.

SVN-013 Function _totalDeposit doesn’t revert when the precision is 0

Total Risk Low	Impact Low	Location connectors/loantoken/LoanTokenLogicStandard.sol
Fixed ✔	Likelihood None	

Description

The function `_totalDeposit` differs in behavior from the original in bZx in that it doesn’t revert when `sourceToDestPrecision` is 0:

```
function _totalDeposit(
    address collateralTokenAddress,
    uint256 collateralTokenSent,
    uint256 loanTokenSent)
    internal
    view
    returns (uint256 totalDeposit)
{
    totalDeposit = loanTokenSent;
    if (collateralTokenSent != 0) {
        (uint256 sourceToDestRate, uint256 sourceToDestPrecision) =
FeedsLike(ProtocolLike(sovrynContractAddress).priceFeeds()).queryRate(
            collateralTokenAddress,
            loanTokenAddress
        );
        if (sourceToDestPrecision != 0) {
            totalDeposit = collateralTokenSent
                .mul(sourceToDestRate)
                .div(sourceToDestPrecision)
                .add(totalDeposit);
        }
    }
}
```

This condition should never happen, but if it happens for any reason it would be safer to revert instead of ignoring it.

Recommendations

Remove the `if` statement and let it revert in `div` if `sourceToDestPrecision` is 0.

9. Disclaimer

The present security audit does not cover the endpoint systems and wallets that communicate with the contracts, nor the general operational security of the company whose contracts have been audited. This document should not be read as investment advice or an offering of tokens.