SMART CONTRACT AUDIT REPORT

for

BENTOS AND SUSHISWAP SETTLEMENT

Prepared By: Shuxiao Wang

Hangzhou, China

December 21, 2020

Document Properties

Client	SushiSwap
Title	Smart Contract Audit Report
Target	BentoBox
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xudong Shao, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 21, 2020	Xuxian Jiang	Final Release
1.0-rc	December 18, 2020	Xuxian Jiang	Release Candidate
0.3	December 17, 2020	Xuxian Jiang	Additional Findings #2
0.2	December 14, 2020	Xuxian Jiang	Additional Findings #1
0.1	December 12, 2020	Xuxian Jiang	Initial Draft

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the **BentoBox** design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About SushiSwap

The SushiSwap lending platform by `SushiSwap` aims to address the limitations in current lending platforms (e.g., `Compound` and `Aave`) by supporting isolated lending pairs, on-chain and off-chain flexible oracles, liquid interest rates, and optimized low gas cost. The isolated lending pairs are unique in containing the risk of current lending platforms that is directly linked to the riskiest asset listed on the platform. (And this risk increases with every extra asset that is added, leading to a very limited choice in assets on most platforms.) Moreover, it supports margin shorting any listed token, which allows for the creation of thousand of lending pairs for any token. The new lending platform also plans to integrate the `flashloan` support in a future version that could bring extra revenue for suppliers. The audited module on `SushiSwap Settlement` also brings the unique much-needed support on limited orders.

The basic information of BentoBox is as follows:

Table 1.1: Basic Information of BentoBox

Item	Description
Issuer	SushiSwap
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 21, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. BentoBox assumes a trusted oracle for each cloned `LendingPair` pair with timely market price feeds and the oracle itself is not part of this audit.

- https://github.com/sushiswap/bentobox.git (c0161d9)
- https://github.com/sushiswap/sushiswap-settlement.git (c01721d)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/sushiswap/bentobox.git (27da15c)
- https://github.com/sushiswap/sushiswap-settlement.git (c01721d)

1.2 About PeckShield

PeckShield Inc. [14] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [13]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [12], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the BentoBox design and implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■■
Low	5	■■■■■
Informational	1	■
Total	9	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 2 medium-severity vulnerabilities, 5 low-severity vulnerabilities, and 1 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Business Logic Issue in swipe()	Business Logic	Fixed
PVE-002	High	Inappropriate Fund Transfer in liquidate()	Business Logic	Fixed
PVE-003	Low	Improved Precision in isSolvent()	Numeric Errors	Fixed
PVE-004	Low	Uninitialized Owner in Deployed LendingPair Clones	Security Features	Fixed
PVE-005	Low	Possible Use of Outdated Exchange Rates	Time and State	Confirmed
PVE-006	Informational	Improved init() in LendingPair Clones	Coding Practices	Confirmed
PVE-007	Low	Improved Sanity Checks Of System/Function Parameters	Coding Practices	Confirmed
PVE-008	Low	Non-ERC20 Compliance of LP Token	Business Logic	Fixed
PVE-009	Medium	Possible Fund Loss From (Permissive) Smart Wallets With Allowances to BentoBox	Business Logic	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Business Logic Issue in swipe()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `LendingPair`
- Category: Business Logic [10]
- CWE subcategory: CWE-837 [5]

Description

With current liquidity pools, `SushiSwap` allows users to trade between a number of supported tokens with a very low slippage. Since its deployment, `SushiSwap` has gained increasing popularity and adoption. In the meantime, we notice that there is always non-trivial possibilities that non-related tokens may be accidentally sent to the pool contract(s). To avoid unnecessary loss of BentoBox users, the BentoBox protocol is designed with the much-needed support of rescuing tokens accidentally sent to the contract. This is a design choice for the benefit of BentoBox users.

During our analysis on the token rescue support, we notice that the current implementation has a business logic issue. To elaborate, we show below the related `swipe()` routine.

```
504     function swipe(IERC20 token) public onlyOwner {
505         if (address(token) == address(0)) {
506             uint256 balanceETH = address(this).balance;
507             if (balanceETH > 0) {
508                 IWETH(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2).withdraw(balanceETH);
509                 (bool success,) = owner.call{value: balanceETH}(new bytes(0));
510                 require(success, "LendingPair: ETH transfer failed");
511             }
512         } else if (address(token) != address(asset) && address(token) != address(
                collateral)) {
513             uint256 balanceAmount = token.balanceOf(address(this));
514             if (balanceAmount > 0) {
515                 (bool success, bytes memory data) = address(token).call(abi.
                        encodeWithSelector(0xa9059cbb, owner, balanceAmount));
516                 require(success && (data.length == 0  abi.decode(data, (bool))), "
                        LendingPair: Transfer failed at ERC20");
```

```
517             }
518         } else {
519             uint256 excessShare = bentoBox.shareOf(token, address(this)).sub(token ==
                    asset ? totalAssetShare : totalCollateralShare);
520             bentoBox.transferShare(token, owner, excessShare);
521         }
522     }
```

Listing 3.1: LendingPair::swipe()

Specifically, if there is an attempt to rescue locked `ETH` in the contract, the execution path at lines 506 – 511 is taken. However, the extra call of `WETH::withdraw()` at line 508 is not part of the logic and should be removed. The presence of this external call will likely revert the rescue attempt.

Recommendation Remove the conflicting `WETH::withdraw()` call in `swipe()`. An example revision is shown below. Note this routine has another related issue that will be elaborated in Section 3.4.

```
504     function swipe(IERC20 token) public onlyOwner {
505         if (address(token) == address(0)) {
506             uint256 balanceETH = address(this).balance;
507             if (balanceETH > 0) {
508                 (bool success,) = owner.call{value: balanceETH}(new bytes(0));
509                 require(success, "LendingPair: ETH transfer failed");
510             }
511         } else if (address(token) != address(asset) && address(token) != address(
                collateral)) {
512             uint256 balanceAmount = token.balanceOf(address(this));
513             if (balanceAmount > 0) {
514                 (bool success, bytes memory data) = address(token).call(abi.
                        encodeWithSelector(0xa9059cbb, owner, balanceAmount));
515                 require(success && (data.length == 0  abi.decode(data, (bool))), "
                        LendingPair: Transfer failed at ERC20");
516             }
517         } else {
518             uint256 excessShare = bentoBox.shareOf(token, address(this)).sub(token ==
                    asset ? totalAssetShare : totalCollateralShare);
519             bentoBox.transferShare(token, owner, excessShare);
520         }
521     }
```

Listing 3.2: LendingPair::swipe()

Status The issue has been fixed in this commit: `0f57eee`.

3.2 Inappropriate Fund Transfer in liquidate()

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High
- Target: `LendingPair`
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [6]

Description

As a lending platform, BentoBox supports the essential functionality in liquidating underwater borrow positions that are not sufficiently backed by collateral. Moreover, the liquidation support features four different types of choices: (1) The first one is the so-called `closed liquidation` using a pre-approved or registered `swapper` for the benefit of the `SushiSwap` LPs; (2) The second one is an `open liquidation` from the liquidator's funds, without making use of any pre-approved `swapper`, but with direct token transfers from the liquidator; (3) The third choice is also an `open liquidation` from the liquidator's funds, without making use of any pre-approved `swapper` either, but with funds already in BentoBox; and (4) The last choice is the so-called `flash liquidation` so that the designated `swapper` gets proceeds first and returns the borrowed amount afterward.

In the following, we show the key code snippet in the `liquidator()` routine. This routine contains a business logic issue in the third choice (lines 455 – 456). Specifically, the tokens are supposed to transfer from the liquidator's BentoBox balance to the `LendingPair` contract itself ($address(this)$). However, it is inappropriately transferred to the liquidator-provided `to` argument: `bentoBox.transferShareFrom(asset, msg.sender, to, allBorrowShare)` (line 455). As a result, the liquidator can claim the underwater borrower's collateral without paying back the borrowed amount.

```
434         if (!open) {
435             // Closed liquidation using a pre-approved swapper for the benefit of the
                    LPs
436             require(masterContract.swappers(swapper), 'LendingPair: Invalid swapper');

438             // Swaps the users' collateral for the borrowed asset
439             uint256 suppliedAmount = bentoBox.transferShareFrom(collateral, address(this
                    ), address(swapper), allCollateralShare);
440             swapper.swap(collateral, asset, suppliedAmount, bentoBox.toAmount(asset,
                    allBorrowShare));
441             uint256 returnedAssetShare = bentoBox.skim(asset);
442             uint256 extraAssetShare = returnedAssetShare.sub(allBorrowShare);

444             // The extra asset gets added to the pool
445             uint256 feeShare = extraAssetShare.mul(protocolFee) / 1e5; // % of profit
                    goes to fee
446             feesPendingShare = feesPendingShare.add(feeShare);
447             totalAssetShare = totalAssetShare.add(extraAssetShare.sub(feeShare));
```

```
448             emit LogAddAsset(address(0), extraAssetShare, 0);
449         } else if (address(swapper) == address(0)) {
450             // Open liquidation directly using the caller's funds, without swapping
                    using token transfers
451             bentoBox.depositShare(asset, msg.sender, allBorrowShare);
452             bentoBox.withdrawShare(collateral, to, allCollateralShare);
453         } else if (address(swapper) == address(1)) {
454             // Open liquidation directly using the caller's funds, without swapping
                    using funds in BentoBox
455             bentoBox.transferShareFrom(asset, msg.sender, to, allBorrowShare);
456             bentoBox.transferShare(collateral, to, allCollateralShare);
457         } else {
458             // Swap using a swapper freely chosen by the caller
459             // Open (flash) liquidation: get proceeds first and provide the borrow after
460             uint256 suppliedAmount = bentoBox.transferShareFrom(collateral, address(this
                    ), address(swapper), allCollateralShare);
461             swapper.swap(collateral, asset, suppliedAmount, bentoBox.toAmount(asset,
                    allBorrowShare));
462             uint256 returnedAssetShare = bentoBox.skim(asset);
463             uint256 extraAsset = returnedAssetShare.sub(allBorrowShare);

465             totalAssetShare = totalAssetShare.add(extraAsset);
466             emit LogAddAsset(address(0), extraAsset, 0);
467         }
```

Listing 3.3: LendingPair :: liquidate ()

Recommendation Properly transfer the borrow amount to the contract itself (e.g., `address(this)`), instead of the liquidator-provided `to` address.

Status The issue has been fixed in this commit: `0f57eee`.

3.3 Improved Precision in isSolvent()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: `LendingPair`
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [2]

Description

Every account in BentoBox needs to avoid being insolvent. If an insolvent situation occurs, the account will go through a liquidation process. Naturally, there is a routine named `isSolvent()` that provides the solvency check. The check is rather straightforward in ensuring the provided collateral is sufficient to maintain the required collaterization ratio with current borrowed amount.

To elaborate, we show below the `isSolvent()` routine. After performing necessary sanity checks on the input arguments, the concrete liquidation requirement is performed at lines 179 – 181. As mentioned earlier, it essentially verifies `collateral * collaterization_rate >= borrow_amount`.

```
170     // Checks if the user is solvent.
171     // Has an option to check if the user is solvent in an open/closed liquidation case.
172     function isSolvent(address user, bool open) public view returns (bool) {
173         // accrue must have already been called!
174         if (userBorrowFraction[user] == 0) return true;
175         if (totalCollateralShare == 0) return false;
177         uint256 borrow = userBorrowFraction[user].mul(totalBorrowShare) /
                totalBorrowFraction;
179         return bentoBox.toAmount(collateral, userCollateralShare[user])
180             .mul(1e18).mul(open ? openCollaterizationRate : closedCollaterizationRate) /
181             exchangeRate / 1e5 >= bentoBox.toAmount(asset, borrow);
182     }
```

Listing 3.4: LendingPair :: isSolvent ()

It is important to note that the lack of `float` support in Solidity may introduce subtle, but troublesome issue: precision loss. One possible precision loss stems from the computation when both multiplication (`mul`) and division (`div`) are involved. Specifically, the compuation at lines 179 – 181 is performed as follows: `bentoBox.toAmount(collateral, userCollateralShare[user]).mul(1e18).mul(open ? openCollaterizationRate : closedCollaterizationRate)/exchangeRate / 1e5 >= bentoBox.toAmount(asset, borrow)`.

A better approach is to avoid any unnecessary division operation that might lead to precision loss. In other words, the comparison of the form `A / B > = C` can be converted into `A >= B * C` under the condition that `B * C` does not introduce any overflow.

Recommendation Avoid unnecessary precision loss due to the lack of floating support in Solidity. An example revision to `isSolvent()` is shown below.

```
170     // Checks if the user is solvent.
171     // Has an option to check if the user is solvent in an open/closed liquidation case.
172     function isSolvent(address user, bool open) public view returns (bool) {
173         // accrue must have already been called!
174         if (userBorrowFraction[user] == 0) return true;
175         if (totalCollateralShare == 0) return false;
177         uint256 borrow = userBorrowFraction[user].mul(totalBorrowShare) /
                totalBorrowFraction;
179         return bentoBox.toAmount(collateral, userCollateralShare[user])
180             .mul(1e13).mul(open ? openCollaterizationRate : closedCollaterizationRate)
181            >= bentoBox.toAmount(asset, borrow).mul(exchangeRate);
182     }
```

Listing 3.5: LendingPair :: isSolvent ()

Status The issue has been fixed in this commit: `0f57eee`.

3.4 Uninitialized Owner in Deployed LendingPair Clones

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LendingPair`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

Ethereum smart contracts are typically immutable by default. Once they are created, there is no way to alter them, effectively acting as an unbreakable contract among participants. In the meantime, there are several scenarios where there is a need to upgrade the contracts, either to add new functionalities or mitigate potential bugs. Or for gas efficiency, multiple minimal proxies can be deployed by sharing the same logic contract as their implementation. An example is the dynamic instantiation of minimal proxies while sharing the same logic `LendingPair` contract in BentoBox.

The minimal proxy support comes with a few caveats. One important caveat is related to the initialization of new contracts that are different from the corresponding logic contract. Due to the inherent requirement of any proxy-based system, no constructors can be used. This means we need to change the constructor of a new contract into a regular function (typically named `initialize()`) that basically executes all the setup logic.

The minimal proxy delegates all calls to its logic contract while keeping all state changes within the proxy. With that, if we re-visit the `swipe()` routine (Section 3.1), the `onlyOwner` modifier deserves special attention. The `owner` state is initialized in the constructor of `Ownable`, which is not executed at all by the minimal proxy. As a result, the `owner` state is in essence uninitialized, hence leading to always `false` by the `onlyOwner` modifier.

```
504     function swipe(IERC20 token) public onlyOwner {
505         if (address(token) == address(0)) {
506             uint256 balanceETH = address(this).balance;
507             if (balanceETH > 0) {
508                 IWETH(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2).withdraw(balanceETH);
509                 (bool success,) = owner.call{value: balanceETH}(new bytes(0));
510                 require(success, "LendingPair: ETH transfer failed");
511             }
512         } else if (address(token) != address(asset) && address(token) != address(
            collateral)) {
513             uint256 balanceAmount = token.balanceOf(address(this));
514             if (balanceAmount > 0) {
515                 (bool success, bytes memory data) = address(token).call(abi.
                        encodeWithSelector(0xa9059cbb, owner, balanceAmount));
```

```
516                 require(success && (data.length == 0  abi.decode(data, (bool))), "
                        LendingPair: Transfer failed at ERC20");
517             }
518         } else {
519             uint256 excessShare = bentoBox.shareOf(token, address(this)).sub(token ==
                        asset ? totalAssetShare : totalCollateralShare);
520             bentoBox.transferShare(token, owner, excessShare);
521         }
522     }
```

Listing 3.6: LendingPair::swipe()

To properly recover lost tokens accidentally sent to the deployed clones of `LendingPair`, there is a need to initialize the `owner` to be `masterContract.owner()`.

Recommendation Properly initialize the `owner` state in the deployed minimal proxies that point to `LendingPair` as their logic contract.

Status The issue has been fixed in this commit: `0f57eee`.

3.5 Possible Use of Outdated Exchange Rates

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LendingPair`
- Category: Time and State [8]
- CWE subcategory: CWE-362 [4]

Description

Throughout the entire SushiSwap protocol, the `LendingPair` contract provides the main entries for borrowers and suppliers. For suppliers, the contract provides `addAsset()/removeAsset()` to allow liquidity providers to supply or un-supply their assets into or out of the pool. For borrowers, the contract provides `addCollateral()/borrow()/repay()` to add collateral into the pool as well as borrow and repay assets. Each operation typically involves the calculation of latest account balance, which necessitates a reliable oracle for real-time price feed.

Our analysis shows that though all these operations rely on latest price feeds, many of them do not always retrieve the latest price feeds. As an example, the `removeCollateral()` operation may not get real-time prices from oracle (see the code snippets below at line 303): the calculation of current solvency is performed with a cached `exchangeRate`. Notice that the use of outdated `exchangeRate` likely lead to inaccurate solvency measurement. In this case, a borrower may be able to borrow more than their collateral is worth. We consider the freshness of these price feeds critical even though their guarantee may introduce additional gas cost.

```
298     // Withdraws a share of collateral of the caller to the specified address
299     function removeCollateral(uint256 share, address to) public {
300         accrue();
301         _removeCollateralShare(msg.sender, share);
302         // Only allow withdrawing if user is solvent (in case of a closed liquidation)
303         require(isSolvent(msg.sender, false), 'LendingPair: user insolvent');
304         bentoBox.withdrawShare(collateral, to, share);
305     }
```

Listing 3.7: LendingPair :: removeCollateral ()

```
170     // Checks if the user is solvent.
171     // Has an option to check if the user is solvent in an open/closed liquidation case.
172     function isSolvent(address user, bool open) public view returns (bool) {
173         // accrue must have already been called!
174         if (userBorrowFraction[user] == 0) return true;
175         if (totalCollateralShare == 0) return false;
176
177         uint256 borrow = userBorrowFraction[user].mul(totalBorrowShare) /
178             totalBorrowFraction;
179
180         return bentoBox.toAmount(collateral, userCollateralShare[user])
181             .mul(1e18).mul(open ? openCollaterizationRate : closedCollaterizationRate) /
182             exchangeRate / 1e5 >= bentoBox.toAmount(asset, borrow);
183     }
```

Listing 3.8: LendingPair :: isSolvent ()

Recommendation Ensure the freshness of price feeds for pool assets. To mitigate possible gas cost, an alternative is to implement the `poke` mechanism in the oracle such that it dynamically notifies the arrival of a new price feed. With that, there is no need to always invoke gas-heavy `updateExchangeRate()` routine before the calculation of account solvency.

Status This is part of protocol design. The team informs us that the idea is that the liquidators (in the case of a closed liquidation, that will be the team) will timely call `updateExchangeRate()`. The team will calculate the actual solvency for each user and liquidate when needed. It has been agreed that the risk here is that someone can borrow more than their collateral is worth, but this may not happen until the price is off by 25%. If the price is off by over 13% or so, it's in the interest of the larger LPs to update the exchange rate.

3.6 Improved init() in LendingPair Clones

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LendingPair`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [1]

Description

The design of BentoBox allows for the creation of many lending pairs for any token. Each pair is associated with two types of tokens: `collateral` and `asset`. Moreover, an `oracle` is bound with the pair that provides the price feeds for position measurement.

During our analysis on its `init()` routine (see the code snippet below), we notice that the routine properly initializes all states of `collateral`, `asset`, and `oracle`. However, it can be improved by further requiring `require(collateral != asset)`. The reason is that the same collateral as the borrowable asset in the proposed model of isolated lending pairs does not make much sense.

```
149     // Serves as the constructor, as clones can't have a regular constructor
150     function init(bytes calldata data) public override {
151         require(address(collateral) == address(0), 'LendingPair: already initialized');
152         (collateral, asset, oracle, oracleData) = abi.decode(data, (IERC20, IERC20,
                IOracle, bytes));
153
154         accrueInfo.interestPerBlock = uint64(startingInterestPerBlock);  // 1% APR, with
                1e18 being 100%
155         updateExchangeRate();
156     }
```

Listing 3.9: LendingPair::init()

Recommendation Validate the given `collateral` and `asset` to ensure they are different. An example revision is shown below:

```
149     // Serves as the constructor, as clones can't have a regular constructor
150     function init(bytes calldata data) public override {
151         require(address(collateral) == address(0), 'LendingPair: already initialized');
152         (collateral, asset, oracle, oracleData) = abi.decode(data, (IERC20, IERC20,
                IOracle, bytes));
153         require(collateral != asset, 'LendingPair: collateral the same as asset')
154
155         accrueInfo.interestPerBlock = uint64(startingInterestPerBlock);  // 1% APR, with
                1e18 being 100%
156         updateExchangeRate();
157     }
```

Listing 3.10: Revised LendingPair::init()

Status This issue is allowed by design as no validity checks are performed on the given tokens. The idea is that though the given pair may be broken or make no sense, it is fine as long as it cannot hurt other users.

3.7 Improved Sanity Checks For System/Function Parameters

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Settlement`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The BentoBox protocol is no exception. Specifically, if we examine the `Settlement` contract, it has defined two inter-related system-wide risk parameters: `feeNumerator` and `feeSplitNumerator`. In the following, we show corresponding routines that allow for their changes.

```
71      // Updates the fee amount and it's split ratio between the relayer and
            feeSplitRecipient
72      function updateFee(uint256 _feeNumerator, uint256 _feeSplitNumerator) public
            onlyOwner {
73          feeNumerator = _feeNumerator;
74          feeSplitNumerator = _feeSplitNumerator;
75      }
```

Listing 3.11: Settlement::updateFee()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `feeNumerator` may use up all funds in the `fillOrder()` operation, hence incurring cost to trading users.

In addition, a number of functions can benefit from more rigorous validation on their arguments. For example, the `_validateArgs()` (see the code below) can be improved by requiring the order's `recipient` is not $address(0)$.

```
191     // Checks if an order is valid - if it contains all the information required
192     function _validateArgs(FillOrderArgs memory args, bytes32 hash) internal view
            returns (bool) {
193         return
194             args.order.maker != address(0) &&
```

```
195             args.order.fromToken != address(0) &&
196             args.order.toToken != address(0) &&
197             args.order.fromToken != args.order.toToken &&
198             args.order.amountIn != uint256(0) &&
199             args.order.amountOutMin != uint256(0) &&
200             args.order.deadline != uint256(0) &&
201             args.order.deadline >= block.timestamp &&
202             args.amountToFillIn > 0 &&
203             args.path.length >= 2 &&
204             args.order.fromToken == args.path[0] &&
205             args.order.toToken == args.path[args.path.length - 1] &&
206             Verifier.verify(args.order.maker, hash, args.order.v, args.order.r, args.
                    order.s);
207     }
```

Listing 3.12: Settlement::_validateArgs()

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status This issue has been confirmed.

3.8 Non-ERC20 Compliance of LP Token

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ERC20
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [6]

Description

In BentoBox, the LendingPair pool implements an ERC20-compliant pool token that represents the ownership of liquidity providers in the shared pool. Accordingly, there is a need for the pool token contract implementation to follow the ERC20 specification. In the following, we examine the list of API functions defined by the ERC20 specification and validate whether there exists any inconsistency or incompatibility in the implementation or the inherent business logic.

Our analysis shows that there are a few ERC20 inconsistency or incompatibility issues found in the audited BentoBox. In particular, according to the ERC20 standard, transfer() and transferFrom() are supposed to revert if the source address does not have enough tokens to spend. However, current implementation simply returns false. In the surrounding two tables, we outline the respective list of basic view-only functions (Table 3.1) and key state-changing functions (Table 3.2) according to the widely-adopted ERC20 specification.

Table 3.1: Basic View-Only Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

Meanwhile, we notice in the transferFrom() routine, there is a common practice that is missing but widely used in other ERC20 contracts. Specifically, when msg.sender = _from, the current transferFrom() implementation disallows the token transfer if msg.sender has not explicitly allows spending from herself yet. A common practice will whitelist this special case and allow transferFrom() if msg.sender = _from even there is no allowance specified. Also, if current allowance is the maximum uint256, there is no need to reduce the allowance as well.

```
30    function transferFrom(address from, address to, uint256 amount) public returns (bool
          success) {
31        if (balanceOf[from] >= amount && allowance[from][msg.sender] >= amount && amount
              > 0 && balanceOf[to] + amount > balanceOf[to]) {
32            balanceOf[from] -= amount;
33            allowance[from][msg.sender] -= amount;
34            balanceOf[to] += amount;
35            emit Transfer(from, to, amount);
36            return true;
37        } else {
38            return false;
39        }
40    }
```

Listing 3.13: ERC20::transferFrom())

Recommendation Be compliant with the widely-accepted ERC20 specification and improve

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	X
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	X
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	X
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approve() event	Is emitted on any successful call to approve()	✓

the `transferFrom()` logic by considering the special case when `msg.sender = _from`.

Status The issue has been fixed in this commit: `0f57eee`.

3.9 Possible Fund Loss From (Permissive) Smart Wallets With Allowances to BentoBox

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `BentoBox`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [6]

Description

Among all core functionalities provided in `BentoBox`, `flashloan` is a disruptive one that allows users to borrow from the reserves within a single transaction, as long as the user returns the borrowed amount plus additional premium. In this section, we report an issue related to the `flashloan` feature. The `flashloan` feature needs to enclose proper checks against potential `reentrancy`.

To elaborate, we show below the code snippet of `flashLoan()` behind the feature.

```
207     // Take out a flash loan
208     function flashLoan(IERC20 token, uint256 amount, address user, bytes calldata params
            ) public checkEntry {
209         uint256 feeAmount = amount.mul(5) / 10000;
210         uint256 returnAmount = amount.add(feeAmount);
211
212         (bool success, bytes memory data) = address(token).call(abi.encodeWithSelector(0
                xa9059cbb, user, amount));
213         require(success && (data.length == 0  abi.decode(data, (bool))), "BentoBox:
                Transfer failed at ERC20");
214         IFlashLoaner(user).executeOperation(token, amount, feeAmount, params);
215         (success, data) = address(token).call(abi.encodeWithSelector(0x23b872dd, user,
                address(this), returnAmount));
216         require(success && (data.length == 0  abi.decode(data, (bool))), "BentoBox:
                TransferFrom failed at ERC20");
217         totalAmount[token] = totalAmount[token].add(feeAmount);
218
219         emit LogFlashLoan(user, token, amount, feeAmount);
220     }
221
222     function flashLoanMultiple(IERC20[] calldata tokens, uint256[] calldata amounts,
            address user, bytes calldata params) public checkEntry {
223         uint256[] memory feeAmounts = new uint256[](tokens.length);
224         uint256[] memory returnAmounts = new uint256[](tokens.length);
225
226         for (uint256 i = 0; i < tokens.length; i++) {
```

```
227             uint256 amount = amounts[i];
228             feeAmounts[i] = amount.mul(5) / 10000;
229             returnAmounts[i] = amount.add(feeAmounts[i]);
230
231             (bool success, bytes memory data) = address(tokens[i]).call(abi.
                    encodeWithSelector(0xa9059cbb, user, amount));
232             require(success && (data.length == 0  abi.decode(data, (bool))), "BentoBox:
                    Transfer failed at ERC20");
233         }
234
235         IFlashLoaner(user).executeOperationMultiple(tokens, amounts, feeAmounts, params)
                ;
236
237         for (uint256 i = 0; i < tokens.length; i++) {
238             (bool success, bytes memory data) = address(tokens[i]).call(abi.
                    encodeWithSelector(0x23b872dd, user, address(this), returnAmounts[i]));
239             require(success && (data.length == 0  abi.decode(data, (bool))), "BentoBox:
                    TransferFrom failed at ERC20");
240             totalAmount[tokens[i]] = totalAmount[tokens[i]].add(feeAmounts[i]);
241
242             emit LogFlashLoan(user, tokens[i], amounts[i], feeAmounts[i]);
243         }
244     }
```

Listing 3.14: BentoBox::flashLoan()

This particular routine implements the `flashloan` feature in a straightforward manner: It firstly transfers the funds to the specified receiver, then invokes the designated operation (`executeOperation` - line 214), and next transfers back the funds from the receiver.

However, our analysis shows that the above logic may be abused to cause fund loss of an innocent user if the user previously specified certain allowances to `BentoBox`. Specifically, if a flashloan is launched by specifying the innocent user as the `user` argument, the `flashLoan()`) execution follows the logic by firstly transferring the loan amount to `user`, invoking `executeOperation()` on the receiver, and then transferring the `returnAmount` (no larger than the allowed spending amount) from the user back to the pool. Note that this flashloan is not initiated by the `user`, who unfortunately pays the premium associated with the flashloan.

The same issue is also applicable to another routine, i.e., `flashLoanMultiple()`. Note the exploitation can be used to directly steal the funds of innocent users, but not for the attacker's benefits. In the meantime, we need to mention that the `executeOperation()` call will be invoked on the given `user`. The compiler will place a sanity check in ensuring the `user` is indeed a contract, hence restricting the attack vector only applicable to contract-based smart wallets. However, current smart wallets may have a fallback routine that could allow the `executeOperation()` call to proceed without being reverted.[1]

[1] An example is those smart wallets in `InstaDApp()`, a popular portal that simplifies the needs for DeFi users.

Recommendation Revisit the design of affected routines in possibly avoiding initiating the `transferFrom()` call from the lending pool. Moreover, the revisited design may validate the `executeOperation()` call so that it is required to successfully transfer back the expected assets, if any.

Status The issue has been fixed by removing the `flashloan` support. We need to clarify that these two vulnerable functions are independently identified (and removed) by the team.

4 Conclusion

In this audit, we have analyzed the BentoBox documentation and implementation. The audited system presents a unique innovation in addressing limitations of current lending platforms. In particular, the proposed approach of isolated lending pairs is especially creative in containing the platform-level risk. We are impressed by the overall design and solid implementation. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

[6] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

[9] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

[11] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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