SUSHI

SushiXSwap and StablePool

Smart Contract Security Review

Version: 1.0

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Sushi smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Sushi smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite on Mainnet Fork, Test Suite on Polygon Fork, Test Suite for `StablePool` , and Test Suite for `StablePool2`).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Sushi smart contracts.

Overview

SushiXSwap is a router contract that batches together transactions to make swaps and transfers across chains. It utilises **Stargate** as a bridging solution and is intergrated with **BentoBox**, **SushiSwap AMM** and **Trident**.

StablePool is a pool contract to facilitate trades between a pair of stablecoins using a constant function for pricing. **StablePoolFactory** enables `MasterDeployer` contract to deploy new `StablePool` contracts.

Security Assessment Summary

This review was conducted on the files hosted on the Sushi repository and were assessed at commit 4888a49 for `SushiXSwap` and commit c993264 for `StablePool` and `StablePoolFactory`.

Specifically, the files in scope are as follows:

- `SushiXSwap.sol`
- `BentoAdapter.sol`
- `StargateAdapter.sol`
- `SushiLegacyAdapter.sol`
- `TokenAdapter.sol`
- `TridentSwapAdapter.sol`
- `StablePool.sol`
- `StablePoolFactory.sol`

An additional target is set for `StablePool.sol` at commit d4346bb which will be referred to as `StablePool2` or `StablePool2.sol` in this document.

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [**?**, **?**].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 19 issues during this assessment. Categorised by their severity:

- Critical: 4 issues.
- High: 1 issue.
- Medium: 4 issues.
- Low: 3 issues.
- Informational: 7 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Sushi smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
SXS-01	Wrong Balance Checked Before Withdrawal	Critical	Open
SXS-02	`safeTransfer` Required for Critical Token Transfer	Critical	Open
SXS-03	`amountOut` Not Converted to `shares`	Critical	Resolved
SXS-04	Amount Conversion Done Twice	Critical	Open
SXS-05	`sgReceive` Could Run Out of Gas	High	Open
SXS-06	Amount Values Within BentoBox Can be Inconsistent	Medium	Open
SXS-07	Check on Stargate Router Address Could Revert	Medium	Open
SXS-08	`sgReceive` Could Send Native Tokens to a Contract	Medium	Open
SXS-09	Incorrect Token Transfer Pattern to `BentoBox`	Medium	Open
SXS-10	Potentially Expired `barFee` and `barFeeTo`	Low	Open
SXS-11	`amount` in `_complexPath()` is a Value in Shares	Low	Open
SXS-12	Accuracy Loss on Pools With Decimals Greater Than 12	Low	Open
SXS-13	Destination Actions Could be Protected	Informational	Open
SXS-14	Input Length Check	Informational	Open
SXS-15	Token Other Than WETH on `_unwrapTransfer()`	Informational	Open
SXS-16	`decimals()` is optional under the `ERC-20` Standard	Informational	Open
SXS-17	Comparing LP Minting Strategy Between `StablePool` and `StablePool2`	Informational	Open
SXS-18	Intentionally Unbalancing Liquidity in `StablePool2`	Informational	Open
SXS-19	Other Miscellaneous Comments	Informational	Open

SXS-01	Wrong Balance Checked Before Withdrawal		
Asset	`SushiXSwap.sol`		
Status	**Open**		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Within the action `ACTION_DST_WITHDRAW_FROM_BENTOBOX`, if the `amount` argument is zero, then the token balance of `address(this)` is assigned to `amount`. However, as this action is for withdrawing from `BentoBox`, the token balance of `address(this)` refers to `SushiXSwap` contract's balance, which is not the desired value to assign. It is the `BentoBox` balance that is needed.

This will only have an impact if both `amount` and `shares` are zero. If this happens, the value of `amount` remains zero and no tokens are transferred out from `BentoBox`. As a result, tokens could be left in `BentoBox` under ownership of the `SushiXSwap` contract, which would allow anyone to withdraw them. Note that, in this instance, the checks in `sgReceive()` would provide no protection as the `payload` transactions would all execute successfully.

Recommendations

If both `amount` and `shares` arguments are zero, check the BentoBox balance using `bentoBox.balanceOf(token,address(this))` and assign it to `shares`.

SXS-02	`safeTransfer` Required for Critical Token Transfer		
Asset	`StargateAdapter.sol`		
Status	**Open**		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The `transfer` function on line [93] of `StargateAdapter.sol` will fail if called with `USDT`, one of the two tokens intended to be used in the function.

This will cause a revert of the call to `sgReceive()`, which will leave the tokens transferred from `Stargate` in the `SushiXSwap` contract on the destination chain, where they can be transferred away freely by any user.

Recommendations

Use `safeTransfer` for the transfer on line [93].

SXS-03	`amountOut` Not Converted to `shares`		
Asset	`StablePool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Function `swap()` swaps `amountIn` of `tokenIn` to `amountOut` of `tokenOut`. To calculate `amountIn`, this function deducts the current balances (`balance0` and `balance1`) with the previous recorded balances (stored in variable `reserve0` and `reserve1`). According to internal functions `_getReservesAndBalances()` and `_balance()`, all of these values (`balance0`, `balance1`, `reserve0`, and `reserve1`) are expressed in `BentoBox`'s `amount` (or elastic). This means, `amountIn` is also in `amount`.

The function `swap()` further uses `amountIn` to calculate `amountOut` through internal function `_getAmountOut()`. It is safe to assume that `amountOut` is also in `amount`, because there is no amount-shares conversion in the `_getAmountOut()` function.

The problem is that `amountOut` is used in function `_transfer()` that requires `shares` and not `amount`. This means that the user may receive more tokens than expected, especially in cases where `BentoBox` receives significant profits from its `strategy` contracts.

Function `getAmountOut()` shows the correct process, where the outcome of function `_getAmountOut()` is converted to share to get `finalAmountOut`.

Recommendations

The testing team recommends converting `amountOut` to `shares` before transferring tokens through function `_transfer()`.

Resolution

The issue has been fixed in commit ec9c147. The `shares` in function `_transfer()` is replaced with (or converted to) `amount`.

SXS-04	Amount Conversion Done Twice		
Asset	`StablePool.sol, StablePool2.sol`		
Status	**Open**		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Variable `reserve0` and `reserve1` store the token pair's balances locally, which are usable for identifying incoming amounts. The values of the variables are taken from `BentoBox` on function `updateReserves()` which calls function `_balance()`. The latter function converts shares into amount through `BentoBox`'s `toAmount()` function, in which the former function stores into `reserve0` and `reserve1`.

However, in some functions such as `_getReservesAndBalances()`, the two variables which are already in amounts are converted to amounts again. The codes in line [**207-208**] utilise `toElastic()` which incorrectly assume `reserve0` and `reserve1` are shares. Similarly on function `_getReserves()`, variable `reserve0` and `reserve1` are converted to amounts through `BentoBox`'s `toAmount()` function. As a result, the token amount calculation becomes highly inaccurate.

A similar issue also occurs in `StablePool2.sol`. The contract `StablePool2` adjusts its reserves before computing liquidity on function `_mintFee()`. This is done twice on functions `_mintFee()` and `_computeLiquidity()` which makes the values inaccurate and causes errors in some cases.

Recommendations

The testing team recommends removing the incorrect conversion in function `_getReservesAndBalances()` and `_getReserves()`.

For `StablePool2`, the testing team recommends removing codes in line [**235-236**], while the code in line [**237**] can be replaced with the following code:

```
computed = _computeLiquidity(_reserve0, _reserve1);
```

Resolution

For `StablePool.sol`, the issue has been fixed in commit ec9c147. Amount conversion codes in function `_getReservesAndBalances()` and `_getReserves()` were deactivated.

SXS-05	`sgReceive` Could Run Out of Gas		
Asset	`StargateAdapter.sol`		
Status	Open		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The development team pointed out that, if the call by `Stargate` to `sgReceive()` were to revert, the tokens transferred from `Stargate` would be left in the `SushiXSwap` contract on the destination chain, where they could be transferred away freely by any user.

One possible condition under which this transaction could revert is if the sequence of actions in the `payload` is long and complex enough that the transaction runs out of gas.

Recommendations

This can be mitigated by carefully checking that large `payload` values are not sent in with insufficient gas.

A more programmatic approach to mitigate this would be to set an explicit gas limit on the `try` call on line [**91**]. If there is an explicit gas limit set in the `try` call, the transaction will execute the `catch` block when this limit is reached. If there is no explicit gas value, then the entire transaction reverts if it runs out of gas.

```
uint256 limit = gasleft() - exitGas;
try
    ISushiXSwap(payable(address(this))).cook{gas: limit}(actions, values, datas)
{} catch (bytes memory) {
    IERC20(_token).safeTransfer(to, amountLD);
}
```

This does waste gas, however, as the `exitGas` state variable would be excess gas sent to every call to `sgReceive`, regardless of whether it ran out of gas or not. This value would need to be high enough to ensure execution of the rest of the `sgReceive()` function.

SXS-06	Amount Values Within BentoBox Can be Inconsistent		
Asset	`SushiXSwap.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

If the actions `ACTION_DST_DEPOSIT_TO_BENTOBOX` and `ACTION_DST_WITHDRAW_FROM_BENTOBOX` are carried out with the same `amount` argument, it is possible for the `ACTION_DST_WITHDRAW_FROM_BENTOBOX` action to revert because the amount of tokens claimed is too high.

This behaviour is caused by the share conversion system within `BentoBox`, which can create imprecisions in token amounts owing to shares being worth more than one token each. `BentoBox` does not always round the imprecision in the same way, and so it is possible that the withdrawal action can request a number of shares that is higher than the share balance stored in `bentoBox.balanceOf`.

The impact of this issue is also mitigated by the checks in `sgReceive()`. As the withrawal action would revert on the destination chain, the `catch` block on line [**93**] should safely transfer the tokens to the user's address.

Recommendations

The following modification to line [**166**] was found to fix the issue:

```
} else if (action == ACTION_DST_WITHDRAW_FROM_BENTOBOX) {
    (
        address token,
        address to,
        uint256 amount,
        uint256 share,
        bool unwrapBento
    ) = abi.decode(
            datas[i],
            (address, address, uint256, uint256, bool)
        );
    if (amount == 0) {
        amount = IERC20(token).balanceOf(address(this));
    } else if (amount > 0 && share == 0 && unwrapBento == true) {
        share = bentoBox.toShare(token, amount, false);
    }

    _transferFromBentoBox(
        token,
        address(this),
        to,
        amount,
        share,
        unwrapBento
    );
```

However, this modification may be undesirable as it will modify all amount calls to the action and could result in unexpected behaviour in a long chain of actions.

Alternatively, be aware of this issue. It can be worked around by using `amount` and `shares` values of zero for `ACTION_DST_WITHDRAW_FROM_BENTOBOX`, so long as the issue SXS-01 is resolved first. User interfaces should implement this practice.

It is also advisable to document the issue well so that third party integrations do not trigger it.

SXS-07	Check on Stargate Router Address Could Revert		
Asset	`StargateAdapter.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The development team pointed out that, if the call by `Stargate` to `sgReceive()` were to revert, the tokens transferred from `Stargate` would be left in the `SushiXSwap` contract on the destination chain, where they could be transferred away freely by any user.

One possible condition under which this transaction could revert is if the `Stargate` router is redeployed, perhaps as part of an upgrade. The `require` on line [80] would then cause the transaction to revert, resulting in a loss of funds.

It is difficult to estimate the likelihood of this issue as it is outside the scope of this review to investigate `Stargate`'s likelihood of redeploying their router. However, whatever their stated policy, there could still be a redeployment and so a risk remains that could result in a loss of user funds.

Recommendations

One possible solution is to remove the `require` on line [80]. This is discussed in more detail in SXS-13.

Alternatively, monitor `Stargate` carefully for any chance that any of their router addresses could change and redeploy this contract if that occurs.

SXS-08	`sgReceive` Could Send Native Tokens to a Contract		
Asset	`StargateAdapter.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The development team pointed out that, if the call by `Stargate` to `sgReceive()` were to revert, the tokens transferred from `Stargate` would be left in the `SushiXSwap` contract on the destination chain, where they could be transferred away freely by any user.

One possible condition under which this transaction could revert is if the user puts a contract with no `receive` or `fallback` function as the `to` address on line [**83**]. In this situation, if any native token balance is present in the `SushiXSwap` contract, then line [**98**] would cause the entire transaction to revert.

A related situation would be if the receiving contract had a `receive` or `fallback` function that uses too much gas, perhaps because of a change in gas fees on the destination chain, and so the `transfer` call would revert.

Recommendations

Use `addr.call{value: x}("")` in place of `transfer` on line [**98**]. This pattern is more generous with gas and also does not revert on failure, which are both properties in line with what this section of code wishes to achieve.

SXS-09	Incorrect Token Transfer Pattern to `BentoBox`		
Asset	`TridentSwapAdapter.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

Function `BentoBox.deposit()` is called on line [28] of `TridentSwapAdapter.sol` to make an initial `BentoBox` deposit to the `Trident` pool. This causes `BentoBox` to attempt a `safeTransferFrom` from the `SushiXSwap` contract. This fails because there is no allowance set for `BentoBox` to spend `SushiXSwap`'s tokens.

This issue will cause a revert. On the source chain, that should only inconvenience the user. On the destination chain, the failed transactions should cause execution to fall into the `catch` block of `sgReceive()`. In each case, there should not be a loss of funds but merely a negative user experience.

Recommendations

Use the "skimming" mechanism of `ACTION_DST_DEPOSIT_TO_BENTOBOX` to transfer the tokens into `BentoBox`: `safeTransfer` to `BentoBox` and then deposit from `BentoBox` to the pool.

SXS-10	Potentially Expired `barFee` and `barFeeTo`		
Asset	`StablePool.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Two variables `barFee` and `barFeeTo` originate from `MasterDeployer` contract. They are used by `StablePool` contract (on function `_mintFee()`) to calculate the system fee and the system fee recipient, respectively. The `StablePool` contract stores these variables locally. These variables are initialised during contract constructor and updateable at anytime using function `updateBarParameters()`.

Technically, the `MasterDeployer`'s owner may change `barFee` or `barFeeTo` at any time. If this occurs, the `StablePool` contract will not be notified and therefore, the stored values in `StablePool` could have been expired when function `_mintFee()` is called. As a result, the fee calculation will be inaccurate or will be sent to the wrong address.

Recommendations

The testing team recommends calling function `updateBarParameters()` inside function `_mintFee()` to update the `barFee` and `barFeeTo` variables. Specifically, the bar parameters update should be done before line [**239**].

SXS-11	`amount` in `_complexPath()` is a Value in Shares		
Asset	`TridentSwapAdapter.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The code in line [65] of `TridentSwapAdapter.sol` references the variable `params.initialPath[i].amount`. This variable is actually measured in shares.

Furthermore, if this action is taking place on the destination chain, it is a concern that the share amounts would have been estimated presumably in the user interface and a significant amount of time would have passed between that moment and this transaction taking place. Based on conversations with the development team, this period could be several minutes: certainly long enough for prices to shift and/or share values to change.

If the calls to `bentoBox.transfer()` attempted to transfer too many shares, they would revert. If they transferred too few, the checks with `params.output[i].minAmount` should cause a revert, and so the user's funds should be protected on both source and destination chains. The impact of this issue is thus only negative user experience.

Recommendations

If the tokens are deposited by the `_complexPath()` function, the share values of those token amounts could be obtained, and these values could then be used in the calls to `bentoBox.transfer()`.

SXS-12	Accuracy Loss on Pools With Decimals Greater Than 12		
Asset	`StablePool.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `adjustedReserve0` and `adjustedReserve1` in function `computeLiquidity` multiply reserved values (`_reserve0` and `_reserve1`) with `1e12` , divided by the tokens' decimals. If this operation is applied to tokens with more than 12 decimals, the value `adjustedReserve0` and `adjustedReserve1` would lose some values.

Consider the following token value: 100.123456789123456789, which is a token with 18 decimals. Multiplying this value with `1e12` and then dividing it with `1e18` would produce 100123456789123.

The adjusted reserves would then be passed to function `_computeLiquidityFromAdjustedBalances()` and `_k()` wherein both values are multiplied.

Recommendations

Make sure this behaviour is understood. The testing team recommends checking the decimals of `token0` and `token1` in the contract `constructor` .

Preventing the use of tokens with decimals greater than 12 will prevent accuracy loss however this issue is partly mitigated by the fact that most tokens have a value such that losses of accuracy below the 12th decimal place have little impact on the real world value of the amounts.

SXS-13	Destination Actions Could be Protected
Asset	`SushiXSwap.sol, StargateAdapter.sol`
Status	**Open**
Rating	Informational

Description

The code in line [**80**] of `StargateAdapter.sol` protects `sgReceive()` from being called by any address except for `stargateRouter`. But, it is questionable what this achieves, considering that all of the actions within `SushiXSwap` can be directly called by any address.

However, there is no reason why destination actions should be so openly accessible. They should only be called through `sgReceive()`, and they are the most dangerous actions.

Recommendations

One of two paths is suggested:

1. **Unrestrict `sgReceive()`** Remove the check on line [**80**] as it is possible to access all the functionality directly and so this check is just as likely to cause an undesirable revert as to protect funds.
2. **Restrict destination actions** If destination actions check that `msg.sender == address(this)`, they would only be accessible through `sgReceive()`. This would provide some measure of protection, although it would still be possible to call them through independent use of the `Stargate` system.

SXS-14	Input Length Check
Asset	`SushiXSwap.sol`
Status	**Open**
Rating	Informational

Description

Function `cook()` requires arrays of input parameters to be of the same length or otherwise the iteration inside the function will likely fail. A transaction fails if `actions.length > values.length` and `actions.length > datas.length`. Unfortunately, there is no check to verify whether `actions.length == values.length == datas.length` or at least `actions.length <= values.length` and `actions.length <= datas.length`.

Recommendations

The testing team recommends that the length of input arrays be checked before executing the function's main operation.

SXS-15	Token Other Than WETH on `_unwrapTransfer()`
Asset	`TokenAdapter.sol`
Status	Open
Rating	Informational

Description

Function `_unwrapTransfer()` unwraps a wrapped ether (WETH) into its native form and transfers it to the sender. This function is called during `ACTION_UNWRAP_AND_TRANSFER`. Since WETH is specific to managing ETH only, it cannot accept any other token. Therefore, if a user enters a token other than WETH in `SushiXSwap._unwrapTransfer()` (line [**181**]), the function will revert.

Recommendations

The testing team recommends only accepting WETH as `token`. The function `_unwrapTransfer()` can be simplified by removing `token` from input parameters. As a replacement, the WETH address can be stored as a constant in the contract, similar to the implementation in `BentoBoxV1` contract. The WETH address can be assigned in the contract's constructor.

Assuming that we have `wethToken` as the WETH contract address, the function can be changed as follows:

```
function _unwrapTransfer(address to) internal {
  IWETH(wethToken).withdraw(IWETH(wethToken).balanceOf(address(this)));
  _transferTokens(IERC20(address(0)), to, address(this).balance);
}
```

SXS-16	`decimals()` is optional under the `ERC-20` Standard
Asset	`StablePool.sol`
Status	Open
Rating	Informational

Description

Although it is common practice to include the `decimals()` function in an `ERC-20` token, the standard does not strictly require it and, in fact, explicitly states:

> OPTIONAL - This method can be used to improve usability, but interfaces and other contracts MUST NOT expect these values to be present.

OpenZeppelin also classifies this attribute as an optional extra.

The constructor of `StablePool.sol` assumes that `decimals()` will always be present. It is possible that an `ERC-20` compliant stablecoin token could exist which this contract would never be able to support, as the constructor would revert on line [**81**] or line [**82**] when it attempts to call `ERC20(_token0/1).decimals()`.

Recommendations

This issue is heavily mitigated by the widespread support for `decimals()` amongst `ERC-20` tokens. However, if full compliance with the standard is desired, the number of decimals could be submitted as an input parameter to the constructor. This has the disadvantage of creating a possible source of input error which might not be immediately obvious until liquidity is added.

SXS-17	Comparing LP Minting Strategy Between `StablePool` and `StablePool2`
Asset	`StablePool.sol, StablePool2.sol`
Status	Open
Rating	Informational

Description

Contract `StablePool` and `StablePool2` are two pool implementations that use different minting strategies. Both are inspired by Solidly Exchange's pool implementation.

During LP minting, the contract `StablePool` computes a new `k` (or liquidity) based on the new token balances, where the added liquidity is calculated by deducting the new `k` with the old `k`.

On the other hand, the contract `StablePool2` computes the added liqudity by simply multiplying amounts with the ratio between LP token supply and the base token amounts. The function `mint()` takes the minimum amount of the two supplied tokens and mints the new liquidity to the liquidity provider.

The advantage of taking the first approach is that the new liquidity takes into account both token balances when the liquidity provider sends unbalanced amounts. The rebalancing of both tokens will be handed over to market mechanism.

While this is not the case for the second approach, where the unbalanced amounts are distributed among all liquidity providers. In other words, a liquidity provider that sends unbalanced amounts will lose the unbalanced amount, while the others will gain the unbalanced amount. The approach taken by `StablePool2` resembles the one implemented by `Solidly`.

Two other differences between the two approaches are:

1. In function `_computeLiquidityFromAdjustedBalances()` where `StablePool` conducts double square-root to the result of `_k()`, while `StablePool2` does not.
2. Contract `StablePool` uses `1e12` to adjust the balances, while `StablePool2` uses `1e16`. As a comparison, `Solidly` uses `1e18`. All are divided by the token's decimal value.

Regarding the maximum amount of liquidity, the contract `StablePool2` has an `Integer overflow` for liquidity over `1e9` on tokens with 18 decimals. On the other hand, `StablePool` contract supports up to `1e12` liquidity on tokens with 18 decimals.

In terms of LP minting accuracy, the difference between `StablePool` and `StablePool2` is deemed insignificant. A liquidity of 100 tokens on 1e9 liquidity (of tokens with 18 decimals) is worth 999.999903295327157597 on `StablePool` and worth 999.999960000099999800 on `StablePool2`, so the difference between the two values is just 0.0005670477.

SXS-18	Intentionally Unbalancing Liquidity in `StablePool2`
Asset	`StablePool2.sol`
Status	Open
Rating	Informational

Description

In `StablePool2.sol`, the calculation of liquidity tokens minted takes place on line [113]:

```
liquidity = StablePoolMath.min((amount0 * _totalSupply) / _reserve0, (amount1 * _totalSupply) / _reserve1);
```

Because the calculation simply takes the lower of two values, each one dependant on the pool's supply of its two tokens, any tokens added in a different ratio from the pool's current reserves will be added to the pool without addition liquidity tokens being awarded.

Users will therefore wish to add tokens to the pool in the ratio of `_reserve0:_reserve1` to maximise their awarded LP tokens, and so their share of the liquidity pool.

However, if the values of `_reserve0` and `_reserve1` are relatively low, it would be possible to frontrun a liquidity depositor with a large swap transaction, followed by a correctly balanced liquidity deposit belonging to the attacker. The victim's liquidity deposit would then land in a heavily unfavourable ratio, resulting in a significantly reduced number of LP tokens being awarded, and thereby significantly increasing the value of all preexisting LP tokens, including those of the attacker.

This form of attack is mitigated by the fact that, at higher values of `_reserve0` and `_reserve1`, it becomes less practical. The necessity for multiple transactions means it cannot be carried out by flash loan.

Recommendations

Make sure this issue is understood. As the code in question has been in use in production on Solidly, it may be that this is a known disadvantage of this form of liquidity calculation.

SXS-19	Other Miscellaneous Comments
Asset	*.sol
Status	Open
Rating	Informational

Description

This section details miscellaneous findings in the `SushiXSwap` contracts.

1. **`General Comments`**

   Lack of Tests and Documentations

   The `SushiXSwap` project lacks tests. While there are some test codes on `StablePool` project, the tests do not cover much of the functionalities of the contracts. The testing team recommends adding rigorous tests to make sure the contracts work as intended.

   It is also worth noting both projects' lack of proper documentation.

2. **`BentoAdapter.sol`**

   Ignored Return Values:

   The best practice is to check return values when they are provided by a function.

   - `_depositToBentoBox()` line [**31**] ignores the return value by `BentoBox.deposit()`.
   - `_transferFromBentoBox()` line [**53**] ignores the return value by `BentoBox.withdraw()`.

3. **`SushiXSwap.sol`**

   3a) **Gas Optimisations:**
   `ACTION_DST_DEPOSIT_TO_BENTOBOX` calculates `amount` if `share` is provided. line [**124**] tests for a zero value of `amount`. If `share` is submitted, `amount` would be needlessly calculated as `share` is used by BentoBox in preference.

   3b) **No events:**
   The development team should be quite certain that they will never want to track the actions of the SushiXSwap contracts and will always be content with monitoring events on the various target contracts.

   3c) **Support for native tokens on destination chain**
   There are comments on line [**124**] wondering whether to support native tokens as the `Stargate` router does not support `value`. It is worth noting that the `Stargate` router can send native tokens as "dust" (to use their terminology) and that native tokens could appear in the `SushiXSwap` desination contract if `ACTION_UNWRAP_AND_TRANSFER` were used to unwrap wrapped native tokens with `SushiXSwap` as the `to` address. Also, native tokens are supported in other destination actions.

   3d) **Minimise Function Access:**
   It is best practice to use the least public permission for a function. The testing team recommends declaring function `cook()` as external. (`StargateAdapter.sol` makes an external call, even though the contract is calling itself.)

   3e) **Coding Style**
   Codes in line [**75-82**] directly call `bentoBox.setMasterContractApproval()`. This coding style is different from other codes in the file that use adapter contracts for calling external functions. To adopt the same style, the development team can write a new function `_setMasterContractApproval()` in `BentoAdapter.sol` that calls `bentoBox.setMasterContractApproval()` through `IBentoBoxMinimal` interface.

4. **`TokenAdapter.sol`**

   4a) **Use of `payable(to).transfer()`:**
   Native tokens sent by this method will cause a revert if they are sent to a receiving contract with no `receive` or `fallback` function. The call will also revert if the `receive` or `fallback` function runs out of gas. Consider the more gas forgiving `addr.call{value: x}("")`, but it may be that the behaviour of `transfer` is what the development team prefers.

5. **`StargateAdapter.sol`**

   Catch all tokens:
   Around line [**93**], an alternative approach for dealing with excess tokens could be to have an empty `catch` block and then test the token balance of `address(this)`. If that balance is above zero, the balance could be sent on to the `to` address. This would perform the same function as the existing `catch` block, but add a little extra check in case tokens are left in the contract for any other reason.

6. **`TridentSwapAdapter.sol`**

   6a) **Complex path comments:**
   line [**57**] implies that tokens are deposited by this function, but they would need to have already been deposited. Similarly, whilst the tokens do ultimately come from the user, in the context of this function as written they come from `BentoBox` deposits belonging to the `SushiXSwap` contract, ie. `address(this)`.

   6b) **Ignored Return Values:**
   It is best practice to check return values when they are provided by a function.
   - `_complexPath()` line [**68**], line [**85**] ignores the return value by `IPool.swap()`.
   - `_complexPath()` line [**99**] ignores the return value by `bentoBox.withdraw()`.

7. **`StablePool.sol`**

   7a) **Unused Variables**
   Variable `price0CumulativeLast` and `price1CumulativeLast` are not used in any part of the contract.

   7b) **Revert Messages and Custom Errors**
   Function `mint()` has two revert messages with literal string on line [**110**] and line [**117**]. These revert messages can be replaced with `error InvalidAmounts()` and `error InsufficientLiquidityMinted()` respectively.
   Similarly, line [**168**] on function `swap()` and line [**345**] on function `getAmountOut()` can be replaced with `error InvalidInputToken()`.

   7c) **Potentially Inaccurate Dev Inline Comment**
   The developer's inline comment on line [**28**] specifies:
   `The curve is applied to shares as well. This pool does not care about the underlying amounts.`
   However, the computation in related functions such as `_getAmountOut()` are in amounts and not shares.

   7d) **Unused Imported Interface**
   Interface `ITridentCallee` is imported but never used in any part of the contract.

   7e) **Empty External Functions**
   The functions `flashSwap()` and `burnSingle()` are implemented as empty functions (for compliance with the `IPool` interface). It is conceivable that a user might call these functions, possibly with token transfers, and it might be prudent, therefore, to include a `revert()` in the body of both functions, as is already the case with `getAmountIn()`.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Appendix A Test Suite on Mainnet Fork

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `brownie` framework was used to perform these tests and the output is given below.

Target: `SushiXSwap.sol` and adapters.

```
test_ACTION_MASTER_CONTRACT_APPROVAL                PASSED  [4%]
test_ACTION_SRC_TOKEN_TRANSFER                      PASSED  [9%]
test_ACTION_DST_WITHDRAW_TOKEN                      PASSED  [14%]
test_ACTION_DST_WITHDRAW_TOKEN_eth                  PASSED  [19%]
test_ACTION_UNWRAP_AND_TRANSFER                     PASSED  [23%]
test_ACTION_SRC_DEPOSIT_TO_BENTOBOX                 PASSED  [28%]
test_ACTION_SRC_DEPOSIT_TO_BENTOBOX_eth             PASSED  [33%]
test_ACTION_SRC_TRANSFER_FROM_BENTOBOX_unwrap       PASSED  [38%]
test_ACTION_SRC_TRANSFER_FROM_BENTOBOX_wrapped      PASSED  [42%]
test_ACTION_DST_DEPOSIT_TO_BENTOBOX                 PASSED  [47%]
test_ACTION_DST_DEPOSIT_TO_BENTOBOX_eth             PASSED  [52%]
test_ACTION_DST_WITHDRAW_FROM_BENTOBOX_amount_issue XFAIL   [57%]
test_ACTION_DST_WITHDRAW_FROM_BENTOBOX_unwrap       PASSED  [61%]
test_ACTION_DST_WITHDRAW_FROM_BENTOBOX_wrap         PASSED  [66%]
test_ACTION_LEGACY_SWAP                             XFAIL   (Exchangera...)
test_ACTION_TRIDENT_SWAP                            PASSED  [76%]
test_ACTION_STARGATE_TELEPORT                       PASSED  [80%]
test_approveToStargateRouter                        PASSED  [85%]
test_sgReceive                                      PASSED  [90%]
test_sgReceive_catch                                XFAIL   (Needto
test_burn_gas                                       PASSED  [100%]
```

Appendix B Test Suite on Polygon Fork

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `brownie` framework was used to perform these tests and the output is given below.

Target: `StablePool.sol` and `StablePoolFactory.sol`.

```
tests/test_sushixswap.py::test_ACTION_TRIDENT_SWAP PASSED                                [ 33%]
tests/test_sushixswap.py::test_ACTION_TRIDENT_SWAP_noapprove XFAIL
    (No approval for token transfer from sushiX to Bento Box)                            [ 66%]
tests/test_sushixswap.py::test_ACTION_TRIDENT_COMPLEX_PATH_SWAP PASSED                   [100%]
```

Appendix C Test Suite for `StablePool`

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `brownie` framework was used to perform these tests and the output is given below.

Target: `StablePool.sol` and `StablePoolFactory.sol`.

```
test_init                          PASSED   [3%]
test_deposit                       PASSED   [6%]
test_mint                          PASSED   [9%]
test_mint_unbalanced               PASSED   [12%]
test_mint_unbalanced_swap          PASSED   [16%]
test_mint_zero                     PASSED   [19%]
test_mint_attack                   PASSED   [22%]
test_burnLiquidity                 PASSED   [25%]
test_liquidity                     PASSED   [29%]
test_liquidity_pbt                 SKIPPED  [32%]
test_burnLiquidity_router          PASSED   [35%]
test_swap                          PASSED   [38%]
test_swap_pbt                      SKIPPED  [41%]
test_new_pool_pbt                  SKIPPED  [45%]
test_new_pool_pbt_loop             SKIPPED  [48%]
test_new_pool                      PASSED   [51%]
test_new_pool_multiswaps           SKIPPED  [54%]
test_new_pool_strategy             PASSED   [58%]
test_getAmountOut                  PASSED   [61%]
test_getAmountOut_pbt              SKIPPED  [64%]
test_getAmountOut_loop             SKIPPED  [67%]
test_getAssets                     PASSED   [70%]
test_bento_shares                  PASSED   [74%]
test_updateBarParameters           PASSED   [77%]
test_new_pool_liquidity_accuracy   PASSED   [80%]
test_constructor                   PASSED   [83%]
test_init                          PASSED   [87%]
test_getDeployData                 PASSED   [90%]
test_getPools                      PASSED   [93%]
test_calculatePoolAddress          PASSED   [96%]
test_deployPool_many               PASSED   [100%]
```

Appendix D Test Suite for `StablePool2`

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `brownie` framework was used to perform these tests and the output is given below.

Target: `StablePool2.sol`.

```
test_init                          PASSED   [3%]
test_deposit                       PASSED   [6%]
test_mint                          PASSED   [10%]
test_mint2                         PASSED   [13%]
test_mint_zero                     PASSED   [17%]
test_burnLiquidity                 PASSED   [20%]
test_mint_unbalanced               PASSED   [24%]
test_liquidity                     PASSED   [27%]
test_burnLiquidity_router          PASSED   [31%]
test_swap                          PASSED   [34%]
test_swap_pbt                      SKIPPED  [37%]
test_new_pool_pbt                  SKIPPED  [41%]
test_new_pool_pbt_loop             SKIPPED  [44%]
test_new_pool                      PASSED   [48%]
test_new_pool_multiswaps           SKIPPED  [51%]
test_new_pool_strategy             PASSED   [55%]
test_getAmountOut                  PASSED   [58%]
test_getAmountOut_pbt              SKIPPED  [62%]
test_getAmountOut_loop             SKIPPED  [65%]
test_getAssets                     PASSED   [68%]
test_bento_shares                  PASSED   [72%]
test_updateBarParameters           PASSED   [75%]
test_new_pool_liquidity_accuracy   PASSED   [79%]
test_constructor                   PASSED   [82%]
test_init                          PASSED   [86%]
test_getDeployData                 PASSED   [89%]
test_getPools                      PASSED   [93%]
test_calculatePoolAddress          PASSED   [96%]
test_deployPool_many               PASSED   [100%]
```

Appendix E Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

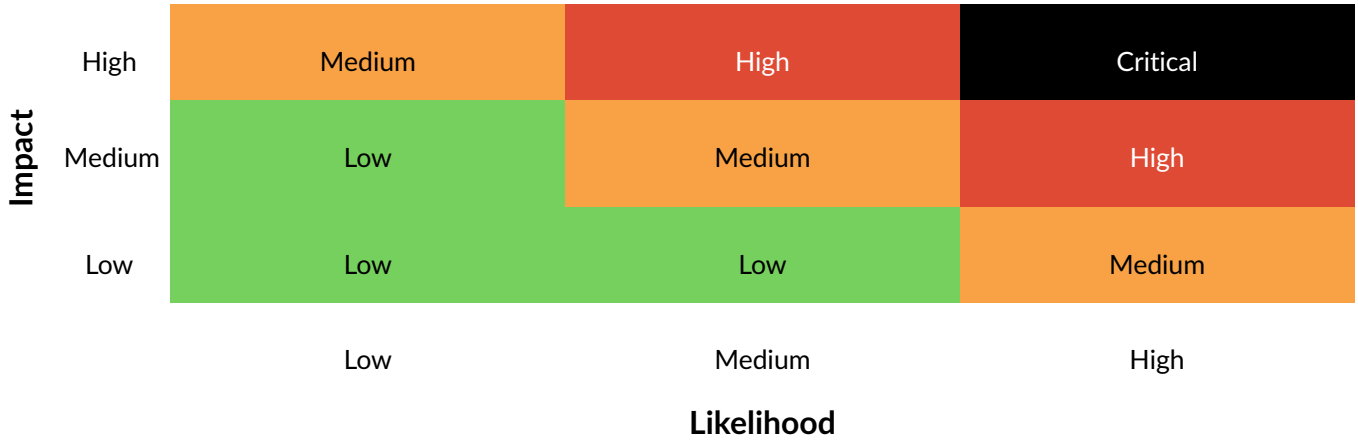

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.