SMART CONTRACT AUDIT REPORT

for

SatoshiSwap

Prepared By: Yiqun Chen

PeckShield

December 26, 2021

Document Properties

Client	SatoshiSwap
Title	Smart Contract Audit Report
Target	SatoshiSwap
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 26, 2021	Jing Wang	Final Release
1.0-rc	December 23, 2021	Jing Wang	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the SatoshiSwap protocol, 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 SatoshiSwap

SatoshiSwap is a decentralized leverage exchange on Binance Smart Chain (BSC). The audited implementation contains four key modules: SatoshiSwap Exchange Module, SatoshiSwap Vault Module, SatoshiSwap Farming Module, and SatoshiSwap Margin Pool. The first three modules are forked from UniswapV2, YearnV2, and SushiSwap MasterChef respectively. The protocol enables traders to readily open leveraged trading positions on BSC trading pairs and enables holders to earn a multitude of passive revenue streams. The basic information of SatoshiSwap is as follows:

Table 1.1: Basic Information of SatoshiSwap

Item	Description
Target	SatoshiSwap
Website	https://satoshiswap.net/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 26, 2021

In the following, we list the reviewed file and the commit hash values used in this audit.

- https://github.com/SatoshiSwap/satoshiswap-protocol.git (a68e4e7)

And here are the commit IDs after all changes for the issues found in the audit have been checked in:

- https://github.com/SatoshiSwap/satoshiswap-protocol.git (a87c4bf)

1.2 About PeckShield

PeckShield Inc. [18] 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 [17]:

- 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 classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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

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) [16], 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 security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment 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 1.4: 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 `SatoshiSwap` 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	2	■■
High	6	■■■■■■
Medium	6	■■■■■■
Low	3	■■■
Informational	1	■
Total	18	

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 that 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, the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 critical-severity vulnerabilities, 6 high-severity vulnerabilities, 6 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 informational recommendation.

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.

Table 2.1: Key SatoshiSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Critical	Necessity of Single-Shot Initialization of VaultStrategy	Initialization and Cleanup	Fixed
PVE-002	High	Business Logic Error in PositionStorage::tradeIn()	Business Logic	Fixed
PVE-003	Medium	Possible Sandwich/MEV Attacks For Reduced Conversion	Time and State	Fixed
PVE-004	Medium	Trust Issue of Admin Keys (Questionable If EOA)	Security Features	Confirmed
PVE-005	High	Several Business Logic Errors in PositionStorage::liquidatePosition()	Business Logic	Fixed
PVE-006	High	Several Business Logic Errors in VPositionStorage::tradeOut()	Business Logic	Fixed
PVE-007	High	Business Logic Error in PositionStorage::_openPosition()	Business Logic	Fixed
PVE-008	Medium	Potential DoS With Vault::withdraw()	Business Logic	Confirmed
PVE-009	Critical	Force Investment Risk in MarginPool::openPosition()	Business Logic	Fixed
PVE-010	Medium	Inconsistent Fee Calculation in MarginPool And PositionStorage	Business Logic	Fixed
PVE-011	High	Several Business Logic Errors in VaultStrategy::prepareReturn()	Business Logic	Fixed
PVE-012	Medium	Possible Costly LP tokens From Improper Vault Initialization	Time and State	Mitigated
PVE-013	Medium	Potential Reentrancy in emergencyWithdraw()	Time and State	Fixed
PVE-014	Informational	Inconsistency Between Document and Implementation	Coding Practices	Fixed
PVE-015	Low	Duplicate Pool Detection and Prevention	Business Logic	Fixed
PVE-016	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Fixed
PVE-017	High	Voting Amplification With Sybil Attacks For SASToken	Business Logic	Fixed
PVE-018	Low	Incompatibility With Deflationary Tokens	Business Logic	Confirmed

3 | Detailed Results

3.1 Necessity of Single-Shot Initialization of VaultStrategy

- ID: PVE-001
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: VaultStrategy
- Category: Initialization and Cleanup [15]
- CWE subcategory: CWE-1188 [3]

Description

The SatoshiSwap protocol has a VaultStrategy contract, which has an initialize() function. This function is used to initialize a number of key parameters, including pool, vault, want, strategist, rewards and keeper. In addition, it also sets up the maximum allowance of the want tokens to the _vault address. To facilitate our discussion, we show below the related code snippet.

```
    function initialize(
        address _pool,
        address _vault,
        address _strategist,
        address _rewards,
        address _keeper
    ) external virtual {
        pool = _pool;
        _initialize(_vault, _strategist, _rewards, _keeper);
    }
```

Listing 3.1: VaultStrategy::initialize()

```
    function _initialize(
        address _vault,
        address _strategist,
        address _rewards,
        address _keeper
    ) internal {
        vault = IVault(_vault);
```

```
        want = IERC20(vault.token());
        want.safeApprove(_vault, uint256(-1)); // Give Vault unlimited access (might
            save gas)
        strategist = _strategist;
        rewards = _rewards;
        keeper = _keeper;

        // initialize variables
        minReportDelay = 0;
        maxReportDelay = 86400;
        profitFactor = 100;
        debtThreshold = 0;

        vault.approve(rewards, uint256(-1)); // Allow rewards to be pulled
    }
```

Listing 3.2: `AbstractBaseStrategy::_initialize()`

Apparently the above logic does not provide the guarantee that the `_initialize()` function can be called only once. What's more, it allows anyone to call the function! A bad actor could call `initialize()` and set the `vault` to his own address, hence transferring all the `want` tokens from the pool. Since multiple initializations could cause critical risk for the entire protocol, we suggest to ensure that the `initialize()` routine may only be called once.

Recommendation Ensure that the `initialize()` function could only be called once during the entire lifetime.

Status This issue has been fixed in the commit: e5ee34c.

3.2 Business Logic Error in PositionStorage::tradeIn()

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High
- Target: `PositionStorage`
- Category: Business Logic [12]
- CWE subcategory: CWE-841 [8]

Description

The `SatoshiSwap` protocol has a `SatoshiSwap Margin Trading Module` which provides the operations of opening a `LONG` or `SHORT` position in accordance with the settings for trading with the `SatoshiSwap Exchange Module`. To facilitate it, the `PositionStorage` contract provides a helper routine, i.e., `tradeIn()`, that is designed to convert user assets into the demanding tokens. To elaborate, we show below the related code snippet.

```
    function tradeIn(PositionLibrary.Trade memory _trade) internal returns (uint256
        swapAmount) {
        ...
        uint256 balanceBefore = IERC20Upgradeable(_trade.quoteToken).balanceOf(address(
            this));

        // execute trade
        ISatoshiRouter(exchangeRouter()).
            swapExactTokensForTokensSupportingFeeOnTransferTokens(
            ...
        );

        uint256 balanceAfter = IERC20Upgradeable(_trade.quoteToken).balanceOf(address(
            this));

        //calculate out amount
        swapAmount = balanceAfter.sub(balanceBefore);

        require(swapAmount > 0, "Margin Pool: Swap failed");

        if (swapAmount < _trade.swapAmount) {
            IERC20Upgradeable(_trade.baseToken).safeTransferFrom(_trade.sender, address(
                this), _trade.swapAmount - swapAmount);
        }
    }
```

Listing 3.3: `PositionStorage::tradeIn()`

We notice this routine swaps `baseToken` to `quoteToken` and checks if the resulting `swapAmount` is smaller than the needed `_trade.swapAmount`. If yes, the routine will transfer the `_trade.swapAmount - swapAmount` amount of `baseToken`, from `_trade.sender` to `PositionStorage`. Since both of the `_trade.swapAmount` and `swapAmount` are amounts of `quoteToken`, it is a logic error to transfer the `_trade.swapAmount - swapAmount` amount of `baseToken` from `_trade.sender`.

Recommendation Correct the above logic error accordingly.

Status This issue has been fixed in the commit: e5ee34c.

3.3 Possible Sandwich/MEV Attacks For Reduced Conversion

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `SatoshiSwap`
- Category: Time and State [14]
- CWE subcategory: CWE-682 [7]

Description

As mentioned in Section 3.2, the `SatoshiSwap` protocol has a `PositionStorage` contract which is used to store the positions and related operations with them. Inside the `PositionStorage` contract, there is a helper routine, i.e., `tradeIn()`, that is designed to convert user assets into the demanding tokens. To elaborate, we show below the related code snippet.

```
    function tradeIn(PositionLibrary.Trade memory _trade) internal returns (uint256 swapAmount) {
        ...
        (uint256 reserve0, uint256 reserve1, ) = pair.getReserves();

        uint256 output = 0;

        //calculate estimate amount out using constant product formula
        if (tokenPair1 == _trade.baseToken) {
            output = SatoshiLibrary.getAmountOut(_trade.input, reserve0, reserve1);
        } else {
            output = SatoshiLibrary.getAmountOut(_trade.input, reserve1, reserve0);
        }

        require(output > 0, "Margin Pool: Satoshi Pool doesnt have reserve");

        //calculate amount with slippage
        uint256 outputWithSlippage = (output.sub(((output.mul(_trade.slippage)).div(denominator()))));

        ...

        // execute trade
        ISatoshiRouter(exchangeRouter()).swapExactTokensForTokensSupportingFeeOnTransferTokens(
            _trade.input,
            outputWithSlippage,
            path,
            address(this),
            block.timestamp.add(delay)
        );
        ...
```

```
    }
```

Listing 3.4: `Satoshiswap::tradeIn()`

We notice the token swap is routed to a router `SatoshiRouter`. And the actual swap operation `swapExactTokensForTokensSupportingFeeOnTransferTokens()` specify an invalid restriction on slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller converted amount. Another routine `tradeOut()` shares the same issue.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user or the virtual account in our case because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above sandwich attack to better protect the interests of protocol users.

Status This issue has been fixed in the commits: e0d16be and 92695e4.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [9]
- CWE subcategory: CWE-287 [4]

Description

In the `SatoshiSwap` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., pool addition, reward adjustment, and parameter setting). It also has the privilege to control or govern the flow of assets managed by this protocol. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    // Set the migrator contract. Can only be called by the owner.
    function setMigrator(IMigratorChef _migrator) public onlyOwner {
        migrator = _migrator;
    }
```

```

    // Migrate lp token to another lp contract. Can be called by anyone. We trust that
        migrator contract is good.
    function migrate(uint256 _pid) public {
        require(address(migrator) != address(0), "migrate: no migrator");
        PoolInfo storage pool = poolInfo[_pid];
        IERC20 lpToken = pool.lpToken;
        uint256 bal = lpToken.balanceOf(address(this));
        lpToken.safeApprove(address(migrator), bal);
        IERC20 newLpToken = migrator.migrate(lpToken);
        require(bal == newLpToken.balanceOf(address(this)), "migrate: bad");
        pool.lpToken = newLpToken;
    }
```

Listing 3.5: `MasterChef::setMigrator()/migrate()`

Specifically, the `MasterChef` contract supports a migration feature that can migrate current pool liquidity to another contract. Notice that the privilege assignment may be necessary and consistent with the protocol design. In the meantime, the extra power to the owner may also be a counter-party risk to the token users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these `onlyOwner` privileges explicit or raising necessary awareness among protocol users.

Moreover, the `management` account could withdraw all `protectedTokens`, which are `want` tokens, from the pool in emergency.

```
//emergency withdraw
function emergencyWithdraw() external management {
    for (uint256 i = 0; i < protectedTokens().length; i++) {
        IERC20Upgradeable(protectedTokens()[i]).safeTransfer(
            IVault(strategy).governance(),
            IERC20Upgradeable(protectedTokens()[i]).balanceOf(address(this))
        );
    }

    IPositionStorage(positionStorage).emergencyWithdraw();
}
```

Listing 3.6: `MarginPool::emergencyWithdraw()`

It is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed with the team. The team clarify they will use a multi-sig account as the `owner`.

3.5 Several Business Logic Errors in PositionStorage::liquidatePosition()

- ID: PVE-005
- Severity: High
- Likelihood: High
- Impact: High
- Target: `PositionStorage`, `MarginPool`
- Category: Business Logic [12]
- CWE subcategory: CWE-841 [8]

Description

In the `SatoshiSwap` protocol, the `PositionStorage` contract also provides a helper routine, `liquidatePosition()`, to trigger the liquidation in case of the liquidation margin is reached. To elaborate, we show below the related code snippet.

```
    function liquidatePosition(
    ) external marginPoolOnly returns (PositionLibrary.SuccessLiquidatePosition memory
        output) {
        //The detailed implementation is removed per the request from the team
        ...
    }
```

Listing 3.7: `PositionStorage::liquidatePosition()`

While analyzing this routine, we notice there are several logic errors. The first one is when this routine refers the `Oracle` to get the price of `position.quoteToken` based on `position.baseToken`. And there is a call of `SatoshiLibrary.sortTokens()` with the input arguments of `position.quoteToken` and `position.baseToken`. The return values `tokenPair1` and `tokenPair2` are passed as the input arguments for `ISatoshiOracle(oracle).getPrice()`. The assumptions of `tokenPair1 == position.quoteToken` and `tokenPair2 == position.baseToken` are not necessarily guaranteed and may be violated if `position.baseToken < position.quoteToke`.

The second one is when this routine evaluates whether the liquidation amount is reached by checking `position.ownedAmount.add(position.userAmount).mul(liquidateMargin()).div(denominator())>= outputAmount`. The `liquidateMargin()).div(denominator()` is directly taken into the multiplication with `position.ownedAmount.add(position.userAmount)`, and compared with `outputAmount`. However, according to the documentation, the logic here should be the opposite way, which takes the `(denominator()-liquidateMargin()).div(denominator())` to multiply with `position.ownedAmount.add(position.userAmount)`.

The third one is the `poolInterestAmount` calculation from `position.ownedAmount > outputAmount`. There is a logic error because when `outputAmount` is smaller than `position.ownedAmount`, there is no interest from this operation, which means the `output.poolInterestAmount` should be 0. However, the current implementation is taking the opposite way. Also, even if we change the `if` condition to the opposite way, the calculation of `output.poolInterestAmount` has another logic error where the `position.ownedAmount.sub(outputAmount)` will always revert as the `position.ownedAmount` is supposed to be smaller than the `outputAmount`.

The fourth one is the `storedBalance` calculation from `storedBalance.sub(output.outputAmount.sub (output.ownedAmount))`. There is a logic error because when `poolInterestAmount` is not larger than 0, it means there is a loss from the `positionStorage::liquidatePosition()`. In this case, the calculation of `output.outputAmount.sub(output.ownedAmount)` will revert as `output.outputAmount` is supposed to be smaller than `output.ownedAmount`.

```
    function liquidatePosition(uint256 _positionId, uint256 _slippage)
        returns (bool success, uint256 reward)
    {
        //The detailed implementation is removed per the request from the team
        ...
    }
```

Listing 3.8: `MarginPool::liquidatePosition()`

Recommendation Correct the logic error mentioned above accordingly.

Status The first issue has been fixed in the commit: aea60cb. The second issue has been fixed in the commit: c6a4aad. The third issue has been fixed in the commit: f0cf155. The fourth issue has been fixed in the commit: 07c070c.

3.6 Several Business Logic Errors in PositionStorage::tradeOut()

- ID: PVE-006
- Severity: High
- Likelihood: High
- Impact: High
- Target: `PositionStorage`
- Category: Business Logic [12]
- CWE subcategory: CWE-841 [8]

Description

As mentioned in Section 3.2, the `SatoshiSwap` protocol has a `SatoshiSwap Margin Trading Module` which provides the operations of opening a `LONG` or `SHORT` position in accordance with the settings for trading with the `SatoshiSwap Exchange Module`. To facilitate it, the `PositionStorage` contract also provides another helper routine, i.e., `tradeOut()`, that is designed to convert the `quoteToken` to demand `baseToken` when a short position is closed. To elaborate, we show below the related code snippet.

```
    function tradeOut(PositionLibrary.Trade memory _trade) internal returns (uint256
        swapAmount) {
        //The detailed implementation is removed per the request from the team
        ...
    }
```

Listing 3.9: `PositionStorage::tradeOut()`

While analyzing this routine, we notice there are several logic errors. The first one is when this routine calculates how much `_trade.baseToken` is needed to do the exchange, `_trade.input.add(_trade.input.mul(_trade.slippage))`. This will route extra `_trade.input.mul(_trade.slippage)` amount of `_trade.baseToken` into the `DEX`.

The second one is the requirement of `_trade.input <= reserve0`. This is problematic because `reserve0` is mapped to the total balance of `token0` from `SatoshiPair`, which may not be `_trade.baseToken`! This will cause the checking of this requirement invalid and further introduce a `denial-of-service` error when performing `tradeOut()`. Note the same issue is also applicable on the requirement of `input[0] <= reserve1`.

The third one is the `swapAmount` which is calculated from `balanceAfter.sub(balanceBefore)`. Semantically, `swapAmount` should be used to represent the swapped amount of `_trade.quoteToken` (`_position.baseToken`). However, as the return value of `tradeOut()`, `swapAmount` is also used as the `profitAmount` in `_closePositionShort()`, which is a logic error. To keep consistency with the implementation in `MarginPool::closePosition()`, we need to subtract the actual `balanceAfter.sub(balanceBefore)` from `_position.swapAmount` to yield the profit.

```
    function closePosition(uint256 _positionId, uint256 _slippage) external nonReentrant
         emergencyShutdown {
        //The detailed implementation is removed per the request from the team
        ...
    }
```

Listing 3.10: `MarginPool::closePosition()`

Recommendation Correct the logic error mentioned above accordingly.

Status The first and second issues have been fixed in the commit: 98c7269. The third issue has been fixed in the commit: 07c070c.

3.7 Business Logic Error in PositionStorage::_openPosition()

- ID: PVE-007
- Severity: High
- Likelihood: High
- Impact: High
- Target: PositionStorage
- Category: Business Logic [12]
- CWE subcategory: CWE-841 [8]

Description

As mentioned in Section 3.2, the SatoshiSwap Margin Trading Module provides the operations of opening a LONG or SHORT position in accordance with the settings for trading with the SatoshiSwap Exchange Module. To facilitate it, the PositionStorage contract provides a helper routine, i.e., _openPosition(), that is designed to accept the user assets and add the leverage to open a position in quoteToken. To elaborate, we show below the related code snippet.

```
    function _openPosition(
    ) internal returns (uint256 ownedAmount, uint256 positionId) {
        //The detailed implementation is removed per the request from the team
        ...
    }
```

Listing 3.11: PositionStorage::_openPosition()

```
    function getAvailablePoolAmount(address _token) internal view returns (uint256) {
        return IGenericMarginPool(pool).getAvailablePoolAmount(_token);
    }
```

Listing 3.12: PositionStorage::getAvailablePoolAmount()

```
    function openPosition(
    ) external nonReentrant emergencyShutdown {
        //The detailed implementation is removed per the request from the team
        ...
    }
```

Listing 3.13: MarginPool::openPosition()

We notice this routine checks the available amount for borrowing before executing the trade to open a position. However, the requirement of amountWithLeverage <= getAvailablePoolAmount(baseToken() obtains the available amount of baseToken from MarginPool, which is a logic error. The reason is that baseTokens are already transferred into positionStorage before the calling of PositionStorage::_openPosition().

Recommendation Correct the logic error mentioned above accordingly.

Status This issue has been fixed in the commit: cbcc182.

3.8 Potential DoS With Vault::withdraw()

- ID: PVE-008
- Severity: High
- Likelihood: High
- Impact: High
- Target: VaultStrategy
- Category: Business Logic [12]
- CWE subcategory: CWE-841 [8]

Description

The SatoshiSwap Vault Module allows users to deposit assets into (and withdraw from) the Vault. When a user intends to withdraw, the withdraw() routine will trigger the related Strategy's withdraw(), which handles the actual withdrawal from the MarginPool. To elaborate, we show below the related code snippet.

```
    function withdraw(uint256 _amountNeeded) external returns (uint256 _loss) {
        require(msg.sender == address(vault), "!vault");
        // Liquidate as much as possible to `want`, up to `_amountNeeded`
        uint256 amountFreed;
        (amountFreed, _loss) = liquidatePosition(_amountNeeded);
        // Send it directly back (NOTE: Using `msg.sender` saves some gas here)
        want.safeTransfer(msg.sender, amountFreed);
        // NOTE: Reinvest anything leftover on next `tend`/`harvest`
    }
```

Listing 3.14: AbstractBaseStrategy::withdraw()

```
    function liquidatePosition(uint256 _amountNeeded) internal override returns (uint256
         _amountFreed, uint256 _loss) {
        if (emergencyExit) {
            ...
        } else {
            uint256 _balance = want.balanceOf(address(this));
            if (_balance >= _amountNeeded) {
                //if we don't set reserve here withdrawer will be sent our full balance
                return (_amountNeeded, 0);
            } else {
                uint256 received = _withdrawSome(_amountNeeded.sub(_balance)).add(
                    _balance);

                return (received, 0);
            }
        }
    }
```

Listing 3.15: VaultStrategy::liquidatePosition()

```
        function _withdrawSome(uint256 _amount) internal returns (uint256) {
            _amount = Math.min(_amount, IGenericMarginPool(pool).nav());

            //dont withdraw dust
            if (_amount < withdrawalThreshold) {
                return 0;
            }

            return IGenericMarginPool(pool).withdraw(_amount);
        }
```

Listing 3.16: VaultStrategy::_withdrawSome()

When analyzing these routines, we notice the `liquidatePosition()` routine will call the `_withdarwSome()` routine, which performs the withdraw from the `MarginPool`. However, if all the funds in the `MarginPool` are borrowed out to open the positions, there is nothing available from `Vault` (because the `liquidatePosition()` routine does not perform a real liquidation). What's more, `liquidatePosition()` never reports a lost on a non-emergency exit so the `Vault` could not be informed of this situation and handle the `debtRatio` accordingly.

Recommendation Properly handle the case when most of the funds are borrowed out from the `MarginPool`.

Status This issue has been confirmed by the team. The team clarifies this is complied with the design.

3.9 Force Investment Risk in MarginPool::openPosition()

- ID: PVE-009
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `MarginPool`
- Category: Security Features [9]
- CWE subcategory: CWE-287 [4]

Description

As mentioned in Section 3.7, the `SatoshiSwap Margin Trading Module` provides the operations of opening a `LONG` or `SHORT` position by a helper routine, `openPosition()`. This routine applies the user configured `_quoteToken` and `_slippage` arguments to open a position. To elaborate, we show below the related code snippet.

```
    function openPosition(
        address _quoteToken,
        PositionLibrary.PositionType _position,
        uint256 _amount,
```

```
        uint256 _leverage,
        uint256 _slippage
    ) external nonReentrant emergencyShutdown {
        ...
        _setBorrowedPoolAmount(_amount.mul(_leverage));

        PositionLibrary.SuccessOpenPosition memory output =
            IPositionStorage(positionStorage).openPosition(msg.sender, _quoteToken,
                _position, _amount, _leverage, _slippage);
        ...
    }
```

Listing 3.17: `MarginPool::openPosition()`

We notice this routine transfers `_amount.mul(_leverage)` amount of `baseToken` to the `PositionStorage` contract and opens the leverage by swapping `baseToken` to `quoteToken` without valid slippage control. A bad actor could add a fake token and manipulate an imbalanced pool to force the `MarginPool` to open a position at an unfavorable exchange rate and do the reversed swap afterwards. Unfortunately, this force investment bug has been exploited in a recent incident (the `veeFinance` hack [1]) that prompts the need of `_quoteToken` validation in `tokenList` and careful `_slippage` management.

Recommendation Correct the above logic error accordingly.

Status This issue has been fixed in the following commits: e0d16be and 118c7f5.

3.10 Inconsistent Fee Calculation Between MarginPool And PositionStorage

- ID: PVE-010
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `MarginPool`, `PositionStorage`
- Category: Business Logics [12]
- CWE subcategory: CWE-841 [8]

Description

In the `MarginPool` contract, the `closePosition()` routine is used to close the position opened by the user and charge the profit fees, if any. To elaborate, we show below the related code snippet of the `supplyFarm()` routine.

```
    function closePosition(uint256 _positionId, uint256 _slippage) external nonReentrant
        emergencyShutdown {
        ...
        PositionLibrary.SuccessClosePosition memory output =
```

```
234             IPositionStorage(positionStorage).closePosition(msg.sender, _positionId,
                    _slippage);
235
236         if (output.positionType == PositionLibrary.PositionType.LONG) {
237             IERC20Upgradeable(token).safeTransferFrom(positionStorage, address(this),
                    output.profitAmount);
238
239             //insurance fee calculation
240
241             uint256 profit = output.profitAmount.sub(output.ownedAmount);
242
243             IERC20Upgradeable(token).safeTransfer(insurance, profit.mul(insuranceFee).
                    div(denominator));
244
245             IERC20Upgradeable(token).safeTransfer(
246                 msg.sender,
247                 output.profitAmount.sub(output.ownedAmount).sub(profit.mul(insuranceFee)
                        .div(denominator)).sub(output.poolInterestAmount)
248             );
249         }
250         ...
251     }
```

Listing 3.18: `MarginPool::closePosition()`

While examining the logic of above function, we notice the fees are divided into two parts: `insuranceFee` and `poolInterestAmount`. However, the fee calculation in `PositionStorage::_closePositionLong()`/`PositionStorage::_closePositionShort()` only counts `poolInterestAmount`. The inconsistent fee calculation between `MarginPool` and `PositionStorage` may revert `MarginPool::closePosition()` when transferring the leftover tokens to `msg.sender` (line 245).

Recommendation Be consistent of fee calculation between `MarginPool` and `PositionStorage`.

Status This issue has been fixed in the commit: a87c4bf.

3.11 Several Business Logic Errors in VaultStrategy::prepareReturn()

- ID: PVE-011
- Severity: High
- Likelihood: High
- Impact: High
- Target: `VaultStrategy`
- Category: Business Logic [12]
- CWE subcategory: CWE-841 [8]

Description

The `SatoshiSwap` protocol has a `SatoshiSwap Vault Module` which accepts deposits from users and provides interaction with strategies. To facilitate it, the `BaseStrategy` contract has defined a set of standard interfaces or methods for `VaultStrategy` to properly interoperate with the vault contract. While examining the `VaultStrategy::prepareReturn()` routine, which is designed to return any realized profits and/or realized losses for the `Vault`'s accounting, we notice there are several logic errors. To elaborate, we show below the related code snippet.

```
    function prepareReturn(uint256 _debtOutstanding)
        internal
        override
        returns (
            uint256 _profit,
            uint256 _loss,
            uint256 _debtPayment
        )
    {
        _profit = 0;
        _loss = 0; //for clarity
        _debtPayment = _debtOutstanding;

        uint256 total = estimatedTotalAssets();

        uint256 looseAssets = want.balanceOf(address(this));
        ...
        uint256 debt = vault.strategies(address(this)).totalDebt;

        if (total > debt) {
            ...
        } else {
            //serious loss should never happen but if it does lets record it accurately
            _loss = debt.sub(total);
            uint256 amountToFree = _loss.add(_debtPayment);

            if (amountToFree > 0 && looseAssets < amountToFree) {
                //withdraw what we can withdraw

                _withdrawSome(amountToFree.sub(looseAssets));
                uint256 newLoose = want.balanceOf(address(this));

                //if we dont have enough money adjust _debtOutstanding and only change
                    profit if needed
                if (newLoose < amountToFree) {
                    if (_loss > newLoose) {
                        _loss = newLoose;
                        _debtPayment = 0;
                    } else {
                        _debtPayment = Math.min(newLoose.sub(_loss), _debtPayment);
                    }
                }
```

```
            }
        }
    }
```

Listing 3.19: `VaultStrategy::prepareReturn()`

The first one is when `total` is smaller than `debt`, which means there is a loss and the routine needs to report it. However, there is a logic error with the calculated `amountToFree`, which should be equal to `_debtPayment` rather than `amountToFree.sub(looseAssets)`.

The second one is also in the case when `total` is smaller than `debt` and the routine needs to report a loss. The `_loss` should not be adjusted based on the relationship between `newLoose` and `amountToFree`.

The third one is the computation of `_debtPayment` in a loss case. The `_debtPayment` should be `newLoose` regardless of whether `newLoose` is smaller than `amountToFree` or not.

Recommendation Correct the above logic error accordingly.

Status This issue has been fixed in the commits: 94bf530 and 342ecf8.

3.12 Possible Costly LPs From Improper Vault Initialization

- ID: PVE-012
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `Vault`
- Category: Time and State [10]
- CWE subcategory: CWE-362 [5]

Description

The `Vault` contract aims to provide incentives so that users can stake and lock their funds in a stake pool. The staking users will get their pro-rata share based on their staked amount. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the share extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `deposit()` routine. This `deposit()` routine is used for participating users to deposit the supported asset (e.g., `token`) and get respective rewards in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
function _issueSharesForAmount(address _to, uint256 _amount) internal returns (uint256
    ) {
    uint256 shares = 0;

    // Issues ‘amount‘ Vault shares to ‘to‘.
    // Shares must be issued prior to taking on new collateral, or
```

```
530      // calculation will be wrong. This means that only *trusted* tokens
531      // (with no capability for exploitative behavior) can be used.
532
533      if (totalSupply() > 0) {
534          //Mint amount of shares based on what the Vault is managing overall
535          //NOTE: if sqrt(token.totalSupply()) > 1e39, this could potentially revert
536          shares = _amount.mul(totalSupply()).div(_totalAssets());
537      } else {
538          shares = _amount;
539      }
540
541      _mint(_to, shares);
542
543      return shares;
544  }
```

Listing 3.20: `valut::_issueSharesForAmount()`

Specifically, when the pool is being initialized, the share value directly takes the value of `shares = amount` (line 538), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `shares = 1 WEI`. With that, the actor can further deposit a huge amount of `token` with the goal of making the share extremely expensive.

An extremely expensive share can be very inconvenient to use as a small number of 1 `Wei` may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Note other routines, i.e., `HandsOn::enter()` and `HandsOnByProxy::enter()`, share the same issue.

Recommendation Revise current execution logic of share calculation to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first deposit to avoid being manipulated.

Status The issue has been mitigated by this commit: `2891a0f`.

3.13 Potential Reentrancy in emergencyWithdraw()

- ID: PVE-013
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MasterChef`, `MasterChefByProxy`
- Category: Time and State [13]
- CWE subcategory: CWE-663 [6]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [20] exploit, and the recent `Uniswap/Lendf.Me` hack [19].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `MasterChef` as an example, the `emergencyWithdraw()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 218) starts before effecting the update on internal states (lines 220−221), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
214 // Withdraw without caring about rewards. EMERGENCY ONLY.
215 function emergencyWithdraw(uint256 _pid) public {
216     PoolInfo storage pool = poolInfo[_pid];
217     UserInfo storage user = userInfo[_pid][msg.sender];
218     pool.lpToken.safeTransfer(address(msg.sender), user.amount);
219     emit EmergencyWithdraw(msg.sender, _pid, user.amount);
220     user.amount = 0;
221     user.rewardDebt = 0;
222 }
```

Listing 3.21: `MasterChef::emergencyWithdraw()`

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`. Note other routines `deposit()` and `withdraw()` share the same issue.

Status The issue has been fixed by this commit: `aea0050`.

3.14 Inconsistency Between Document and Implementation

- ID: PVE-014
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SatoshiPair`
- Category: Coding Practices [11]
- CWE subcategory: CWE-1041 [2]

Description

There is a misleading comment embedded in the `SatoshiPair` contract, which brings unnecessary hurdles to understand and/or maintain the software.

The preceding function summary indicates that this function is supposed to mint liquidity *"equivalent to 1/6th of the growth in sqrt(k)"* However, the implementation logic (line $113-116$) indicates the minted liquidity should be equal to 1/3 of the growth in `sqrt(k)`.

```
103     // if fee is on, mint liquidity equivalent to 1/6th of the growth in sqrt(k)
104     function _mintFee(uint112 _reserve0, uint112 _reserve1) private returns (bool feeOn) {
105         address feeTo = ISatoshiFactory(factory).feeTo();
106         feeOn = feeTo != address(0);
107         uint256 _kLast = kLast; // gas savings
108         if (feeOn) {
109             if (_kLast != 0) {
110                 uint256 rootK = Math.sqrt(uint256(_reserve0).mul(_reserve1));
111                 uint256 rootKLast = Math.sqrt(_kLast);
112                 if (rootK > rootKLast) {
113                     uint256 numerator = totalSupply.mul(rootK.sub(rootKLast));
114                     uint256 denominator = rootK.mul(2).add(rootKLast); //0.15% liquidity fee, 0.005% protocol - 1/3
115                     uint256 liquidity = numerator / denominator;
116                     if (liquidity > 0) _mint(feeTo, liquidity);
117                 }
118             }
119         } else if (_kLast != 0) {
120             kLast = 0;
121         }
122     }
```

Listing 3.22: `BoxswapPair::_mintFee()`

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status This issue has been fixed in the commit: e5ee34c.

3.15 Duplicate Pool Detection and Prevention

- ID: PVE-015
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MasterChef`, `MasterChefByProxy`
- Category: Business Logic [12]
- CWE subcategory: CWE-841 [8]

Description

The `SatoshiSwap` protocol provides incentive mechanisms that reward the staking of supported assets with certain reward tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its `allocPoint*100%/totalAllocPoint` share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded tokens and more can be scheduled for addition (via a proper governance procedure). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in `add()`, whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool but with a duplicate token from being added. Though it is a privileged interface (protected with the modifier `onlyOwner`), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function add(
        uint256 _allocPoint,
        IERC20 _lpToken,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({lpToken: _lpToken, allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock, accSASPerShare: 0}));
    }
```

Listing 3.23: `MasterChef::add()`

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

```
    function checkPoolDuplicate(IERC20 _lpToken) public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            require(poolInfo[_pid].lpToken != _lpToken, "add: existing pool?");
        }
    }

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you
         do.
    function add(
        uint256 _allocPoint,
        IERC20 _lpToken,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        checkPoolDuplicate(_lpToken);
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({lpToken: _lpToken, allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock, accSASPerShare: 0}));
    }
```

Listing 3.24: Revised `MasterChef::add()`

We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers.

Status The issue has been fixed by this commit: `e5ee34c`.

3.16 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-016
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MasterChef`, `MasterChefByProxy`
- Category: Business Logic [12]
- CWE subcategory: CWE-841 [8]

Description

As mentioned in Section 3.16, the `SatoshiSwap` protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. When analyzing the pool weight update routine `set()`, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```
    // Update the given pool's sas allocation point. Can only be called by the owner.
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.25: `MasterChef::set()`

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed.

```
    // Update the given pool's sas allocation point. Can only be called by the owner.
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) public onlyOwner {
        massUpdatePools();
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.26: `MasterChef::set()`

Status The issue has been fixed by this commit: `e5ee34c`.

3.17 Voting Amplification With Sybil Attacks

- ID: PVE-017
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: SASToken, SASTokenByProxy
- Category: Business Logic [12]
- CWE subcategory: CWE-841 [8]

Description

The SAS tokens can be used for governance in allowing for users to cast and record the votes. Moreover, the SASToken contract allows for dynamic delegation of a voter to another, though the delegation is not transitive. When a submitted proposal is being tallied, the number of votes are counted via getPriorVotes().

Our analysis shows that the current governance functionality is vulnerable to a new type of so-called Sybil attacks. For elaboration, let's assume at the very beginning there is a malicious actor named Malice, who owns 100 SAS tokens. Malice has an accomplice named Trudy who currently has 0 balance of SASs. This Sybil attack can be launched as follows:

```
    function _delegate(address delegator, address delegatee) internal {
        address currentDelegate = _delegates[delegator];
        uint256 delegatorBalance = balanceOf(delegator); // balance of underlying SASs (
            not scaled);
        _delegates[delegator] = delegatee;

        emit DelegateChanged(delegator, currentDelegate, delegatee);

        _moveDelegates(currentDelegate, delegatee, delegatorBalance);
    }

    function _moveDelegates(
        address srcRep,
        address dstRep,
        uint256 amount
    ) internal {
        if (srcRep != dstRep && amount > 0) {
            if (srcRep != address(0)) {
                // decrease old representative
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].
                    votes : 0;
                uint256 srcRepNew = srcRepOld.sub(amount);
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
            }

            if (dstRep != address(0)) {
                // increase new representative
```

```
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].
                    votes : 0;
                uint256 dstRepNew = dstRepOld.add(amount);
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
            }
        }
    }
```

Listing 3.27: SASToken.sol

1. Malice initially delegates the voting to Trudy. Right after the initial delegation, Trudy can have 100 votes if he chooses to cast the vote.

2. Malice transfers the full 100 balance to M_1 who also delegates the voting to Trudy. Right after this delegation, Trudy can have 200 votes if he chooses to cast the vote. The reason is that the SASToken contract's transfer() does NOT _moveDelegates() together. In other words, even now Malice has 0 balance, the initial delegation (of Malice) to Trudy will not be affected, therefore Trudy still retains the voting power of 100 SASs. When M_1 delegates to Trudy, since M_1 now has 100 SASs, Trudy will get additional 100 votes, totaling 200 votes.

3. We can repeat by transferring M_i's 100 SAS balance to M_{i+1} who also delegates the votes to Trudy. Every iteration will essentially add 100 voting power to Trudy. In other words, we can effectively amplify the voting powers of Trudy arbitrarily with new accounts created and iterated!

Recommendation To mitigate, it is necessary to accompany every single transfer() and transferFrom() with the _moveDelegates() so that the voting power of the sender's delegate will be moved to the destination's delegate. By doing so, we can effectively mitigate the above Sybil attacks.

Status The issue has been fixed by this commit: e5ee34c.

3.18 Incompatibility with Deflationary Tokens

- ID: PVE-018
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [12]
- CWE subcategory: CWE-841 [8]

Description

In the `SatoshiSwap` protocol, the `Masterchef` contract is designed to take users' assets and deliver rewards depending on their share. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdraw()`, allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract using the `safeTransferFrom()` or `safeTransfer()` routine to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    // Deposit LP tokens to MasterChef for sas allocation.
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user.amount.mul(pool.accSASPerShare).div(1e12).sub(user.
                rewardDebt);
            safeSASTransfer(msg.sender, pending);
        }
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accSASPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }

    // Withdraw LP tokens from MasterChef.
    function withdraw(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(_pid);
        uint256 pending = user.amount.mul(pool.accSASPerShare).div(1e12).sub(user.
            rewardDebt);
        safeSASTransfer(msg.sender, pending);
        user.amount = user.amount.sub(_amount);
        user.rewardDebt = user.amount.mul(pool.accSASPerShare).div(1e12);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
```

```
        emit Withdraw(msg.sender, _pid, _amount);
    }
```

Listing 3.28: `MasterChef::deposit()and MasterChef::withdraw()`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every `transfer()` or `transferFrom()`. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the pool and affects protocol-wide operation and maintenance.

Specially, if we take a look at the `updatePool()` routine. This routine calculates `pool.accTokenPerShare` via dividing `cakeReward` by `lpSupply`, where the `lpSupply` is derived from `balanceOf(address(this))` (line 172). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may give a big `pool.accTokenPerShare` as the final result, which dramatically inflates the pool's reward.

```
166     // Update reward variables of the given pool to be up-to-date.
167     function updatePool(uint256 _pid) public {
168         PoolInfo storage pool = poolInfo[_pid];
169         if (block.number <= pool.lastRewardBlock) {
170             return;
171         }
172         uint256 lpSupply = pool.lpToken.balanceOf(address(this));
173         if (lpSupply == 0) {
174             pool.lastRewardBlock = block.number;
175             return;
176         }
177         uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
178         uint256 sasReward = multiplier.mul(sasPerBlock).mul(pool.allocPoint).div(
            totalAllocPoint);
179         sas.mint(devaddr, sasReward.div(10));
180         sas.mint(address(this), sasReward);
181         pool.accSASPerShare = pool.accSASPerShare.add(sasReward.mul(1e12).div(lpSupply))
            ;
182         pool.lastRewardBlock = block.number;
183     }
```

Listing 3.29: `Masterchef::updatePool()`

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `SatoshiSwap` for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary. Note another contract `Vault` shares the same issue.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate.

Status The issue has been confirmed by the team. And the team clarifies that they don't support deflationary token in this version and they will support deflationary tokens in next version.

4 Conclusion

In this audit, we have analyzed the `SatoshiSwap` design and implementation. The audited implementation contains four modules including `SatoshiSwap Exchange Module`, `SatoshiSwap Vault Module`, `SatoshiSwap Farming Module`, and `SatoshiSwap Margin Pool`. The current code base is well organized. Those identified issues are confirmed or addressed.

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] Vee Finance. The Main Cause of Vee Finance Attack. https://veefi.medium.com/the-main-cause-of-vee-finance-attack-7a8475085ec5.

[2] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[3] MITRE. CWE-1188: Insecure Default Initialization of Resource. https://cwe.mitre.org/data/definitions/1188.html.

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

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

[6] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[7] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

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

[13] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[14] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[15] MITRE. CWE CATEGORY: Initialization and Cleanup Errors. https://cwe.mitre.org/data/definitions/452.html.

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

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

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

[19] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[20] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.