SMART CONTRACT AUDIT REPORT

for

BabySwap

Prepared By: Yiqun Chen

PeckShield

November 30, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	November 30, 2021	Jing Wang	Final Release
1.0-rc	November 30, 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 BabySwap 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 BabySwap

BabySwap is a decentralized automated market making (AMM) protocol for newborn projects on Binance Smart Chain (BSC). It emphasizes the concept of "Baby is the Future". The trader could find potential baby projects on BabySwap early and accompany them to grow up to 'rock stars' through trading, farming and bottling. The project could get the best support on BabySwap, including growth funds, arbitrage supports, entertaining activities, resource connections, friendly displays, etc.

The basic information of the BabySwap protocol is as follows:

Table 1.1: Basic Information of The BabySwap Protocol

Item	Description
Name	BabySwap
Website	https://home.babyswap.finance/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 30, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/babyswap/baby-swap-contract (cac289b)

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

- https://github.com/babyswap/baby-swap-contract (92fb1e0)

Note the following files are NOT included in this audit scope: `ReBuy.sol`, `VBabyToken.sol`, `LotteryRewardPool.sol`, `SousChef.sol`, `BabyRouter01.sol`, `BabyRouter02.sol`, `DecimalMath.sol` and `BabyLibrary.sol`.

1.2 About PeckShield

PeckShield Inc. [13] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [12]:

- 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.

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) [11], 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 1.3: The Full List of Check Items

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

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

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

2 | Findings

2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	1	■
Medium	4	■ ■ ■ ■
Low	6	■ ■ ■ ■ ■ ■
Informational	0	
Total	11	

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, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerabilities, 4 medium-severity vulnerabilities, and 6 low-severity vulnerabilities.

Table 2.1: Key BabySwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-002	High	Voting Amplification With Sybil Attacks	Business Logics	Confirmed
PVE-003	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	Confirmed
PVE-004	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Confirmed
PVE-005	Medium	Improved Reward Calculation In SwapMining::takerWithdraw()	Time and State	Confirmed
PVE-006	Low	Suggested Adherence of Checks-Effects-Interactions Pattern	Time and State	Confirmed
PVE-007	Low	Possible Costly LPs From Improper AutoBabyPool Initialization	Time and State	Confirmed
PVE-008	Low	Incompatibility with Deflationary Tokens	Business Logics	Confirmed
PVE-009	Low	Duplicate Pool Detection and Prevention	Business Logic	Confirmed
PVE-010	Medium	Improved Deletion Logic In Bottle::withdraw()	Business Logic	Confirmed
PVE-011	Medium	Improved Logic For Same Transaction Deposit() And Withdraw() Handling In AutoBabyPool	Time And State	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Trust Issue of Admin Keys

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [1]

Description

In the BabySwap protocol, there is a special administrative account, i.e., owner. This owner account plays a critical role in governing and regulating the system-wide operations (e.g., minting tokens , setting various parameters, and migrating current liquidity). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

To elaborate, we show below the transferBabyTokenOwnerShip() and transferSyrupOwnerShip() routines in the MasterChefTimelock contract. These two routines transfer the privileged accounts to newOwner_, which could mint any amount of tokens to anyone, as long as the totalSupply() is smaller than the maxSupply. What's more, these two functions are bypassing the delay of Timelock, which means the ownership transaction process would be faster than expected and pose counter-party risk to the users.

```
    function transferBabyTokenOwnerShip(address newOwner_) external onlyAdmin {
        masterChef.transferBabyTokenOwnerShip(newOwner_);
    }
```

Listing 3.1: MasterChefTimelock::transferBabyTokenOwnerShip()

```
    function transferSyrupOwnerShip(address newOwner_) external onlyAdmin {
        masterChef.transferSyrupOwnerShip(newOwner_);
```

```
    }
```

Listing 3.2: `MasterChefTimelock::transferSyrupOwnerShip()`

```
    function mintFor(address _to, uint256 _amount) public onlyOwner {
        _mint(_to, _amount);
        require(totalSupply() <= maxSupply, "reach max supply");
        _moveDelegates(address(0), _delegates[_to], _amount);
    }
```

Listing 3.3: `BabyToken::mintFor()`

```
    function mint(address _to, uint256 _amount) public onlyOwner {
        _mint(_to, _amount);
        _moveDelegates(address(0), _delegates[_to], _amount);
    }
```

Listing 3.4: `SyrupBar::mint()`

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 mitigated by the following changeset: `92fb1e0`.

3.2 Voting Amplification With Sybil Attacks

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Multiple Contracts`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

The `BABY` tokens can be used for governance in allowing for users to cast and record the votes. Moreover, the `BABY` 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 BABY tokens. Malice has an accomplice named Trudy who currently has 0 balance of BABYs. 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 CAKEs (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 3.5: BabyToken.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 BABY 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 BABYs. When M_1 delegates to Trudy, since M_1 now has 100 BABYs, Trudy will get additional 100 votes, totaling 200 votes.

3. We can repeat by transferring M_i's 100 BABY 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! Note the same issue also exists on SYRUP.

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. Since the contract is already deployed, it is safe and acceptable to deploy another contract for governance, and use the current one for other ERC-20 functions only. A cleaner solution would be to migrate the current contract to a new one with the suggested fix, but the migration effort may be costly.

Status The issue has been confirmed by the team. The team clarifies that the voting feature is currently not used.

3.3 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Time and State [10]
- CWE subcategory: CWE-682 [4]

Description

The BabySwapFee contract has a helper routine, i.e., doHardwork(), that is designed to split the fees to different components. It has a rather straightforward logic in removing liquidity and allowing router to transfer the funds by calling swapExactTokensForTokensSupportingFeeOnTransferTokens() to actually perform the intended token swap.

```
    function doHardwork(address[] calldata pairs, uint minAmount) external {
        require(msg.sender == caller, "illegal caller");
        for (uint i = 0; i < pairs.length; i ++) {
            IBabyPair pair = IBabyPair(pairs[i]);
            if (pair.token0() != USDT && pair.token1() != USDT) {
                continue;
            }
```

```
            uint balance = pair.balanceOf(address(this));
            if (balance == 0) {
                continue;
            }
            if (balance < minAmount) {
                continue;
            }
            balance = transferToVault(pair, balance);
            address token = pair.token0() != USDT ? pair.token0() : pair.token1();
            pair.approve(address(router), balance);
            router.removeLiquidity(
                token,
                USDT,
                balance,
                0,
                0,
                address(this),
                block.timestamp
            );
            address[] memory path = new address[](2);
            path[0] = token;path[1] = USDT;
            balance = IBEP20(token).balanceOf(address(this));
            IBEP20(token).approve(address(router), balance);
            router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
                balance,
                0,
                path,
                address(this),
                block.timestamp
            );
        }
    }
```

Listing 3.6: `BabySwapFee::doHardwork()`

To elaborate, we show above the `doHardwork()` routine. We notice the remove liquidity and token swap are routed to `router` and the actual removal or swap operation via `removeLiquidity()` or `swapExactTokensForTokensSupportingFeeOnTransferTokens()` essentially do not specify any restriction (with `amountOutMin=0`) on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

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 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 front-running attack to better protect the interests of farming users.

Status The issue has been confirmed by the team. And the team clarifies that since there won't be large amount of trading, so it won't trigger the sandwich attack.

3.4 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MasterChef`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `BabySwap` protocol has a `MasterChef` contract that 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 CAKE allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 prevAllocPoint = poolInfo[_pid].allocPoint;
        poolInfo[_pid].allocPoint = _allocPoint;
        if (prevAllocPoint != _allocPoint) {
            totalAllocPoint = totalAllocPoint.sub(prevAllocPoint).add(_allocPoint);
            updateStakingPool();
        }
    }
```

Listing 3.7: 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. Note other routine `SwapMining::setPair()` shares the same issue.

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 CAKE allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
        massUpdatePools();
        uint256 prevAllocPoint = poolInfo[_pid].allocPoint;
        poolInfo[_pid].allocPoint = _allocPoint;
        if (prevAllocPoint != _allocPoint) {
            totalAllocPoint = totalAllocPoint.sub(prevAllocPoint).add(_allocPoint);
            updateStakingPool();
        }
    }
```

Listing 3.8: MasterChef::set()

Status The issue has been confirmed by the team. And the team clarifies that if there is an error in the configuration of the pool, the forced update will cause the related operation fail also.

3.5 Sandwich Attacks For SwapMining Rewards

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `SwapMining`
- Category: Time and State [10]
- CWE subcategory: CWE-682 [4]

Description

The `SwapMining` protocol is forked from `MDX` and provides incentives when users make a swap. The protocol provides rewards based on the swapping amount for the supported assets. While examining the reward calculation with the given swap amount, we notice the `takerWithdraw()` may be sandwiched by two swaps with reversed paths. To elaborate, we show the `SwapMining::takerWithdraw()` routine.

```
// The user withdraws all the transaction rewards of the pool
function takerWithdraw() public {
    uint256 userSub;
    uint256 length = poolInfo.length;
    for (uint256 pid = 0; pid < length; ++pid) {
        PoolInfo storage pool = poolInfo[pid];
        UserInfo storage user = userInfo[pid][msg.sender];
        if (user.quantity > 0) {
            mint(pid);
```

```
            // The reward held by the user in this pool
            uint256 userReward = pool.allocMdxAmount.mul(user.quantity).div(pool.quantity);
            pool.quantity = pool.quantity.sub(user.quantity);
            pool.allocMdxAmount = pool.allocMdxAmount.sub(userReward);
            user.quantity = 0;
            user.blockNumber = block.number;
            userSub = userSub.add(userReward);
        }
    }
    if (userSub <= 0) {
        return;
    }
    console.log(userSub);
    babyToken.transfer(msg.sender, userSub);
}
```

Listing 3.9: `SwapMining::takerWithdraw()`

Our analysis shows that the given `SwapMining` contract may be exploited by flashloans. Specifically, a bad actor could accumulate the `user.quantity` by making a flashloans of swapping token A to token B. After taking the rewards from the `takerWithdraw()` routine, the bad actor could take a reversed swap and make profits again. The bad actor could repeat the above steps to make profits as long as the value of the reward is larger than swap fees.

```
// swapMining only router
  function swap(address account, address input, address output, uint256 amount) public onlyRouter returns (bool) {
      require(account != address(0), "SwapMining: taker swap account is the zero address");
      require(input != address(0), "SwapMining: taker swap input is the zero address");
      require(output != address(0), "SwapMining: taker swap output is the zero address");

      if (poolLength() <= 0) {
          return false;
      }

      if (!isWhitelist(input)  !isWhitelist(output)) {
          return false;
      }

      address pair = BabyLibrary.pairFor(address(factory), input, output);
      PoolInfo storage pool = poolInfo[pairOfPid[pair]];
      // If it does not exist or the allocPoint is 0 then return
      if (pool.pair != pair  pool.allocPoint <= 0) {
          return false;
      }

      uint256 quantity = getQuantity(output, amount, targetToken);
      if (quantity <= 0) {
```

```
            return false;
        }

        mint(pairOfPid[pair]);

        pool.quantity = pool.quantity.add(quantity);
        pool.totalQuantity = pool.totalQuantity.add(quantity);
        UserInfo storage user = userInfo[pairOfPid[pair]][account];
        user.quantity = user.quantity.add(quantity);
        user.blockNumber = block.number;
        return true;
    }
```

Listing 3.10: SwapMining::swap()

Recommendation Develop an effective mitigation to the above sandwich attack to ensure the proper computation and dissemination of `swapMining` reward.

Status The issue has been confirmed by the team. And the team clarifies that they want to keep the `SwapMining` part as a free market. It will give users who provide LP more rewards, and won't be harmful for the project.

3.6 Suggested Adherence of Checks-Effects-Interactions Pattern

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterChef`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [3]

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` [15] exploit, and the recent `Uniswap/Lendf.Me` hack [14].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `MasterChef` as an example, the `withdraw()` function (see the code snippet below) is provided to withdraw funds from the pool. However, the invocation of an external contract to transfer token requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 258) starts before effecting the update on internal states (lines 264), 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 very same `withdraw()` function.

```
248     function withdraw(uint256 _pid, uint256 _amount) public {

250         require (_pid != 0, 'withdraw CAKE by unstaking');
251         PoolInfo storage pool = poolInfo[_pid];
252         UserInfo storage user = userInfo[_pid][msg.sender];
253         require(user.amount >= _amount, "withdraw: not good");

255         updatePool(_pid);
256         uint256 pending = user.amount.mul(pool.accCakePerShare).div(1e12).sub(user.
                rewardDebt);
257         if(pending > 0) {
258             safeCakeTransfer(msg.sender, pending);
259         }
260         if(_amount > 0) {
261             user.amount = user.amount.sub(_amount);
262             pool.lpToken.safeTransfer(address(msg.sender), _amount);
263         }
264         user.rewardDebt = user.amount.mul(pool.accCakePerShare).div(1e12);
265         emit Withdraw(msg.sender, _pid, _amount);
266     }
```

Listing 3.11: `MasterChef::withdraw()`

Note other routines `BnbStaking::deposit()` and `BnbStaking::withdraw()` and `MasterChef::deposit()` share the same issue.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status The issue has been confirmed by the team.

3.7 Possible Costly LPs From Improper AutoBabyPool Initialization

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: AutoBabyPool
- Category: Time and State [7]
- CWE subcategory: CWE-362 [2]

Description

The AutoBabyPool 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., BABY) 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 deposit(uint256 _amount)
        external
        whenNotPaused
        notContract
        nonReentrant("deposit")
    {
        require(_amount > 0, "Nothing to deposit");

        uint256 pool = balanceOf();
        token.safeTransferFrom(msg.sender, address(this), _amount);
        uint256 currentShares = 0;
        if (totalShares != 0) {
            currentShares = (_amount.mul(totalShares)).div(pool);
        } else {
            currentShares = _amount;
        }
        UserInfo storage user = userInfo[msg.sender];

        user.shares = user.shares.add(currentShares);
        user.lastDepositedTime = block.timestamp;

        totalShares = totalShares.add(currentShares);

        user.cakeAtLastUserAction = user.shares.mul(balanceOf()).div(
            totalShares
        );
        user.lastUserActionTime = block.timestamp;
```

```
170
171         _earn();
172
173         emit Deposit(msg.sender, _amount, currentShares, block.timestamp);
174     }
```

Listing 3.12: `AutoBabyPool::deposit()`

Specifically, when the pool is being initialized, the share value directly takes the value of `currentShares = _amount` (line 157), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `currentShares = 1 WEI`. With that, the actor can further deposit a huge amount of `BABY` 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.

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 confirmed by the team. And the team clarifies that the related contract has been deployed and initialized.

3.8 Incompatibility with Deflationary Tokens

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `Multiple Contracts`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In the `BabySwap` 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 CAKE allocation.
    function deposit(uint256 _pid, uint256 _amount) public {

        require (_pid != 0, 'deposit CAKE by staking');

        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user.amount.mul(pool.accCakePerShare).div(1e12).sub(user.rewardDebt);
            if(pending > 0) {
                safeCakeTransfer(msg.sender, pending);
            }
        }
        if (_amount > 0) {
            pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
            user.amount = user.amount.add(_amount);
        }
        user.rewardDebt = user.amount.mul(pool.accCakePerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }

    // Withdraw LP tokens from MasterChef.
    function withdraw(uint256 _pid, uint256 _amount) public {

        require (_pid != 0, 'withdraw CAKE by unstaking');
        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.accCakePerShare).div(1e12).sub(user.rewardDebt);
        if(pending > 0) {
            safeCakeTransfer(msg.sender, pending);
        }
        if(_amount > 0) {
            user.amount = user.amount.sub(_amount);
            pool.lpToken.safeTransfer(address(msg.sender), _amount);
        }
        user.rewardDebt = user.amount.mul(pool.accCakePerShare).div(1e12);
        emit Withdraw(msg.sender, _pid, _amount);
```

```
    }
```

Listing 3.13: `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 213). 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.

```
207     // Update reward variables of the given pool to be up-to-date.
208     function updatePool(uint256 _pid) public {
209         PoolInfo storage pool = poolInfo[_pid];
210         if (block.number <= pool.lastRewardBlock) {
211             return;
212         }
213         uint256 lpSupply = pool.lpToken.balanceOf(address(this));
214         if (lpSupply == 0) {
215             pool.lastRewardBlock = block.number;
216             return;
217         }
218         uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
219         uint256 cakeReward = multiplier.mul(cakePerBlock).mul(pool.allocPoint).div(
                totalAllocPoint);
220         cake.mintFor(address(syrup), cakeReward);
221         pool.accCakePerShare = pool.accCakePerShare.add(cakeReward.mul(1e12).div(
                lpSupply));
222         pool.lastRewardBlock = block.number;
223     }
```

Listing 3.14: `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 `BabySwap` for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary. Note other contracts including PoolFactory, Holdstake and IDO share 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 the project party should ensure that the token is a standard ERC20 token. If the token would be a deflationary one, the project should add the pool contract to the whitelist to avoid the additional charges.

3.9 Duplicate Pool Detection and Prevention

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Multiple Contracts
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The MasterChef 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 privileged function). 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, IBEP20 _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,
            accCakePerShare: 0
        }));
        updateStakingPool();
    }
```

Listing 3.15: `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. Note the `Holdstake`, `NFTFarm`, `ILO` ,`SwapMining` contracts share the same issue.

```
    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, IBEP20 _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,
            accCakePerShare: 0
        }));
        updateStakingPool();
    }
```

Listing 3.16: 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 confirmed by the team.

3.10 Improved Deletion Logic In Bottle::withdraw()

- ID: PVE-010
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Bottle
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the BabySwap protocol, the Bottle contract provides incentive mechanisms that reward the voting of supported _voteId with certain reward tokens. It allows the user to start a pool and vote for the supporting _voteId by adding funds during the pool's voting schedule. While reviewing the implementation of the pool deletion logic, we notice the new pool may fail to be created because of the wrongly deleted old pool. To elaborate, we show below the withdraw() function in the Bottle contract.

```
function withdraw(uint256 _voteId, address _for) external nonReentrant {
    createPool();
    //require(currentVoteId <= 4  _voteId >= currentVoteId - 4, "illegal voteId");
    PoolInfo memory _pool = poolInfo[_voteId];
    require(_pool.avaliable, "illegal voteId");
    require(block.timestamp > _pool.unlockAt, "not the right time");
    UserInfo memory _userInfo = userInfo[_voteId][msg.sender][_for];
    require (_userInfo.amount > 0, "illegal amount");

    //uint _pending = masterChef.pendingCake(0, address(this));
    uint256 balanceBefore = babyToken.balanceOf(address(this));
    masterChef.leaveStaking(0);
    uint256 balanceAfter = babyToken.balanceOf(address(this));
    uint256 _pending = balanceAfter.sub(balanceBefore);
    uint _totalShares = totalShares;
    if (_pending > 0 && _totalShares > 0) {
        accBabyPerShare = accBabyPerShare.add(_pending.mul(RATIO).div(_totalShares));
    }

    uint _userPending = _userInfo.pending.add(_userInfo.amount.mul(accBabyPerShare).div(
        RATIO).sub(_userInfo.rewardDebt));
    uint _totalPending = _userPending.add(_userInfo.amount);

    if (_totalPending >= _pending) {
        masterChef.leaveStaking(_totalPending.sub(_pending));
    } else {
        //masterChef.leaveStaking(0);
        babyToken.approve(address(masterChef), _pending.sub(_totalPending));
```

```
184         masterChef.enterStaking(_pending.sub(_totalPending));
185     }
186 
187     //if (_totalPending > 0) {
188         SafeBEP20.safeTransfer(babyToken, msg.sender, _totalPending);
189     //}
190 
191     if (_userPending > 0) {
192         emit Claim(_voteId, msg.sender, _for, _userPending);
193     }
194 
195     totalShares = _totalShares.sub(_userInfo.amount);
196     poolInfo[_voteId].totalAmount = _pool.totalAmount.sub(_userInfo.amount);
197 
198     delete userInfo[_voteId][msg.sender][_for];
199     if (poolInfo[_voteId].totalAmount == 0) {
200         delete poolInfo[_voteId];
201         emit DeleteVote(_voteId);
202     }
203     emit Withdraw(_voteId, msg.sender, _for, _userInfo.amount);
204 }
```

Listing 3.17: `Bottle::withdraw()`

The deletion of the current pool, i.e., `delete userInfo[_voteId][msg.sender][_for]` (line 198), may be performed when `block.timestamp < _currentPool.finishAt`. In other words, the old pool may be deleted before a new pool created. In this case, `_currentPool.finishAt` will give a 0 value and the new created pool's voting schedule is invalid.

Recommendation Improve the pool deletion logic in `Bottle::withdraw()`.

Status The issue has been confirmed by the team.

3.11 Improved Logic For Same Transaction Deposit() And Withdraw() Handling In AutoBabyPool

- ID: PVE-011
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: AutoBabyPool
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.10, the AutoBabyPool contract aims to provide incentives so that users can stake and lock their funds in a stake pool. If we compare with other staking contracts, the BABY staked into this snack pool will be automatically compounded (or reinvested). An unstaking fee applies when the user unstakes within 3 days of staking.

While examining the logic, we notice an exploiter could deposit() and withdraw() in one transaction and earn BABYs as long as _earn() delivers enough rewards. To elaborate, we show below the deposit() and withdraw() routines from AutoBabyPool.

```
    function deposit(uint256 _amount)
        external
        whenNotPaused
        notContract
        nonReentrant("deposit")
    {
        require(_amount > 0, "Nothing to deposit");

        uint256 pool = balanceOf();
        token.safeTransferFrom(msg.sender, address(this), _amount);
        uint256 currentShares = 0;
        if (totalShares != 0) {
            currentShares = (_amount.mul(totalShares)).div(pool);
        } else {
            currentShares = _amount;
        }
        UserInfo storage user = userInfo[msg.sender];

        user.shares = user.shares.add(currentShares);
        user.lastDepositedTime = block.timestamp;

        totalShares = totalShares.add(currentShares);

        user.cakeAtLastUserAction = user.shares.mul(balanceOf()).div(
            totalShares
        );
        user.lastUserActionTime = block.timestamp;
```

```
        _earn();

        emit Deposit(msg.sender, _amount, currentShares, block.timestamp);
    }
```

Listing 3.18: AutoBabyPool::deposit()

```
    function withdraw(uint256 _shares)
        public
        notContract
        nonReentrant("withdraw")
    {
        UserInfo storage user = userInfo[msg.sender];
        require(_shares > 0, "Nothing to withdraw");
        require(_shares <= user.shares, "Withdraw amount exceeds balance");

        uint256 currentAmount = (balanceOf().mul(_shares)).div(totalShares);
        user.shares = user.shares.sub(_shares);
        totalShares = totalShares.sub(_shares);

        uint256 bal = available();
        if (bal < currentAmount) {
            uint256 balWithdraw = currentAmount.sub(bal);
            IMasterChef(masterchef).leaveStaking(balWithdraw);
            uint256 balAfter = available();
            uint256 diff = balAfter.sub(bal);
            if (diff < balWithdraw) {
                currentAmount = bal.add(diff);
            }
        }

        if (block.timestamp < user.lastDepositedTime.add(withdrawFeePeriod)) {
            uint256 currentWithdrawFee = currentAmount.mul(withdrawFee).div(
                10000
            );
            token.safeTransfer(treasury, currentWithdrawFee);
            currentAmount = currentAmount.sub(currentWithdrawFee);
        }

        if (user.shares > 0) {
            user.cakeAtLastUserAction = user.shares.mul(balanceOf()).div(
                totalShares
            );
        } else {
            user.cakeAtLastUserAction = 0;
        }

        user.lastUserActionTime = block.timestamp;

        token.safeTransfer(msg.sender, currentAmount);

        emit Withdraw(msg.sender, currentAmount, _shares);
```

```
404     }
```

Listing 3.19: `AutoBabyPool::withdraw()`

Our analysis shows that a bad actor makes a profit as long as there is enough reward accumulated from `MasterChef` (e.g. being idle for a long time without `harvest()` or other actions). The calling of `_earn()` (line 171) from `deposit()` could give more rewards than the `currentWithdrawFee` (line 384), thus covering the cost for the one transaction `deposit()` and `withdraw()`.

Recommendation Improve the logic for same transaction `deposit()` and `withdraw()` handling in the `AutoBabyPool` contract.

Status The issue has been confirmed by the team.

4 Conclusion

In this audit, we have analyzed the BabySwap protocol design and implementation. The BabySwap protocol is a decentralized AMM on Binance Smart Chain (BSC) with the emphasized concept of "Baby is the Future" where providing better support for newborn projects. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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