
rev1.0

Prepared for
Mound

Prepared by
Theori

**July 30th, 2021**

Theori

# Table of Contents

# Executive Summary

Starting on June 21, 2021, Theori assessed the Pancake Bunny smart contracts and relayer for six weeks. We focused on identifying issues that put deposits at risk, correctness issues, and attacks that could mint excess reward tokens. We found two critical issues in the relayer that could result in a loss of funds from the relay contracts. We also found two other issues in the relayer that may result in a loss of funds but whose impacts are limited. The three correctness issues were judged to have minimal security impact.

During the audit of PancakeBunny, there was an attack on the PolygonBunny smart contracts. While PolygonBunny is outside of the scope of this audit, we examined the root cause of this attack, whether PancakeBunny has similar issues, and possible mitigations. We did not identify any immediately critical issues, but the Venus pool may be susceptible to the same attack in some situations. PancakeBunny should reconsider its method of inflation when minting reward tokens.

# Scope

The audit covered smart contracts for PancakeBunny on the Binance Smart Chain and the relayer scripts to implement the ETH-backed pools. Only code within the following source code repositories was reviewed:

- PancakeBunny-finance/Bunny
    - 43f4718f01135521750d7408115306d0cf3d74e1
- mound/public_bunny-casper
    - ee80d27840d38ad724585c87542e95eab574e95c
- mound/public_bunny-firebase
    - 857604f82e4e7ed215086fe2787abd7bc28a61ff

# Discussion

Recently, PancakeBunny and its sister product, PolygonBunny, have suffered two attacks. Both attacks resulted in an attacker acquiring a large amount of BUNNY reward tokens which were immediately sold for profit at the expense of current BUNNY holders. In neither attack were user assets at risk. This speaks to the safety of the PancakeBunny vaults.

The goal of this audit was a comprehensive audit of the PancakeBunny smart contracts and an audit of the beta ETH pool and relayer. As recent attacks focused on minting reward tokens, we also focused our attention on this attack surface. We also paid extra attention on the beta ETH pool and relayer as it is significantly less tested than the rest of PancakeBunny.

We now will review the two recent attacks, their root causes, and the fixes. We also discuss the risk of sandwich attacks when swapping tokens on the BSC blockchain.

## Oracle attack on PancakeBunny

The flash loan attack on May 19, 2021, exploited the price oracle that was being used by PancakeBunny to calculate the number of reward tokens to mint in exchange for the performance fee. Instead of using an off-chain oracle or an oracle that was resistant to manipulation, PancakeBunny was using only the current exchange rates on PancakeSwap to calculate asset prices. This is known to be incredibly dangerous due to oracle manipulation attacks.

PancakeBunny fixed the issue by using a trusted price feed from ChainLink to calculate the price of assets. For tokens that are not supported by ChainLink, PancakeBunny provides their own reference price data. The only downside to this fix is that the price data can become out-of-date. The timestamp of the ChainLink feed is currently ignored, so a problem with the ChainLink oracle may result in stale data. If the reference price data timestamp is more than a day old, a price of zero is returned instead. Stale data is probably better than returning zero but reverting the current transaction would be preferred.

We reviewed the fixed PriceCalculator smart contracts and did not find any major issues. The only price that can be manipulated by the attacker is the price of a liquidity provider (LP) token. It is unclear if this is an issue that needs to be fixed, but we discuss it further in the Recommendations section so that it is a known risk.

## Excess profit attack on PolygonBunny

A second attack on July 16, 2021, exploited PolygonBunny, a product similar to PancakeBunny running on Polygon. Since it is impossible to manipulate the price oracle, the attacker instead manipulated the performance fee. Normally this would not be an issue

because it would not be profitable to increase the performance fee, however both PancakeBunny and PolygonBunny mint bonus BUNNY reward tokens based on the performance fee. At the time of the attack on PolygonBunny, the performance fee is converted to BUNNY at a rate of 1.5x.

While the cause of the vulnerability in PolygonBunny was the manipulation of the profit of a pool, it would not be profitable if there was no inflation of the BUNNY reward tokens. For every 1 unit of asset that the attacker deposited as "profit," the attacker earned a bonus of 15% in BUNNY tokens (30% performance fee x 1.5x inflation rate). We discuss in the Recommendations section possible mitigations to prevent exploitation of the minter, one of which is to simply stop minting BUNNY tokens in exchange for the performance fee.

The announcement of the attack indicated that BSC was not affected by this attack. As this audit of PancakeBunny was ongoing during the attack, we re-reviewed the BSC vaults to ensure that their profits could not be manipulated. We did not find any immediate issues, but we identified two related issues: the price of LP tokens might be manipulated to increase profit and the borrowings of a Venus pool could be repaid to increase the profit. We discuss these two findings in the Findings and Recommendations sections.

## Sandwich attacks

During the audit, it was noticed that the calls to PancakeSwap to swap tokens did not specify a minimum output amount. The documentation[1] for Uniswap details the safety considerations when swapping tokens and specifically details the risk of a sandwich attack. It is recommended that smart contracts use a price oracle to determine a reasonable minimum output amount to prevent this type of attack.

The mitigating factor for PancakeBunny is that its token swaps are small relative to the liquidity of the PancakeSwap pools, because most of the token swaps for PancakeBunny are due to a harvest call that swaps reward tokens. An attacker would require a large amount of capital to manipulate the price and would not make a profit most of the time. As the amount of rewards increases so does the potential profit for an attacker, though it is likely that the pools would also have more liquidity.

However, PancakeBunny is not the only project that has not addressed this risk. Most similar projects also swap tokens without specifying a minimum output amount. As PancakeBunny is in line with current industry practices, we recommend that this risk is addressed but do not consider it to be a security issue.

[1] https://docs.uniswap.org/protocol/V2/guides/smart-contract-integration/trading-from-a-smart-contract

# Findings

These are the potential issues that may have correctness and/or security impacts. Most of the issue have already been remediated by the PancakeBunny team.

## Summary

| # | ID | Title | Severity |
|---|---|---|---|
| 1 | THE-BUNNY-001 | Failure to reset state after relayer withdraw | Critical |
| 2 | THE-BUNNY-002 | Missing authorization check on relayer getReward | Critical |
| 3 | THE-BUNNY-003 | Attacker can drain funds by spamming relayer | High |
| 4 | THE-BUNNY-004 | Relayer does not properly check for chain reorg | High |
| 5 | THE-BUNNY-005 | Relayer liqudation fails to check queue state | Low |
| 6 | THE-BUNNY-006 | Incorrect rate calculation in TripleSlopeModel | Low |
| 7 | THE-BUNNY-007 | Total supply is modified before notifyDeposited | Low |
| 8 | THE-BUNNY-008 | Attacker can manipulate profit of Venus pool | Medium |

## Issue #1: Failure to reset state after relayer withdraw

| ID | Summary | Severity |
|---|---|---|
| THE-BUNNY-001 | Due to a failure to reset the user's withdraw history, a user can repeatedly withdraw profits from the relayer | Critical |

A withdraw from the ETH pools is a multi-step process:

1. User submits a withdraw request to the relayer via HTTP API
2. Relayer calls VaultRelayer's withdrawBySig
    a. User's assets are converted to BNB
    b. Profit and loss are added to withdrawnHistories[pool][account]
3. Relayer calls VaultRelayer's withdrawnHistoryOf to get profit and loss and passes these values to VaultCollateral's unlockCollateral (on Ethereum)
    a. VaultCollateral subtracts loss from user's collateral
    b. VaultCollateral adds profit to _realizedProfit[account]
4. Relayer calls VaultRelayer's completeWithdraw
5. User calls VaultCollateral's removeCollateral to retrieve collateral and profit

During this process the values in withdrawnHistories are never cleared. If the user submits another withdraw, the relayer will get the same values for profit and loss in step 3 and the profit will be added again to the user's realized profit. By repeatedly submitting a withdraw, a user can multiply their profits.

The ETH pool feature is in a closed beta which limited the possibility for an attacker to exploit this issue. We were able to demonstrate the issue with a whitelisted account.

### Fix

PancakeBunny fixed the issue by clearing withdrawnHistories[pool][account] in VaultRelayer's completeWithdraw. They also added a check in _withdraw to validate that completeWithdraw has been called.

```
function _withdraw(address pool, address account) private {
    require(!withdrawing[pool][account], "VaultRelayer: withdrawing must be
complete");
```

```
    if (VaultRelayInternal(pool).balanceOf(account) == 0) return;
    withdrawing[pool][account] = true;

    (uint flipAmount, uint cakeAmount) = _withdrawInternal(pool, account);
    uint bnbAmount = _zapOutToBNB(pool, flipAmount, cakeAmount);
    (uint profitInETH, uint lossInETH) = bank.repayAll{value : bnbAmount}(pool,
account);

    PoolConstant.RelayWithdrawn storage history = withdrawnHistories[pool][account];
    history.pool = pool;
    history.account = account;
    history.profitInETH = history.profitInETH.add(profitInETH);
    history.lossInETH = history.lossInETH.add(lossInETH);

    if (profitInETH > lossInETH) {
        bank.bridgeETH(BRIDGE, profitInETH.sub(lossInETH));
    }
    emit Withdrawn(pool, account, profitInETH, lossInETH);
}

function completeWithdraw(address pool, address account) external onlyWhitelisted {
    delete withdrawing[pool][account];
    delete withdrawnHistories[pool][account];
}
```

## Issue #2: Missing authorization check on relayer getReward

| ID | Summary | Severity |
|---|---|---|
| THE-BUNNY-002 | The public getReward function in VaultRelayInternal is missing the onlyRelayer modifier allowing an attacker to steal rewards | Critical |

The getReward function in VaultRelayInternal is missing the onlyRelayer modifier:

```
function getReward(address _to) public nonReentrant updateReward(msg.sender) {
    uint reward = rewards[_to];
    if (reward > 0) {
        rewards[_to] = 0;
        uint before = IBEP20(CAKE).balanceOf(address(this));
        _rewardsToken.withdraw(reward);
        uint cakeBalance = IBEP20(CAKE).balanceOf(address(this)).sub(before);

        IBEP20(CAKE).safeTransfer(msg.sender, cakeBalance);
        emit ProfitPaid(_to, cakeBalance, 0);
    }
}
```

This function would allow an attacker to withdraw the CAKE rewards for any account. The only difficulty for an attacker is that updateReward is called with msg.sender instead of _to. An attacker would need to wait until an account deposits additional assets before calling getReward to steal the rewards.

The ETH pool feature is in a closed beta which limited the possibility for an attacker to exploit this issue.

### Fix

PancakeBunny fixed the issue by adding the onlyRelayer modifier. The updateReward modifier was also fixed to update the rewards for _to instead of msg.sender. The git commit is public at https://github.com/PancakeBunny-finance/Bunny/commit/a5982d987ca8cbbe6891612622ee9ecb41c35186.

```
function getReward(address _to) public nonReentrant updateReward(_to) onlyRelayer {
    uint reward = rewards[_to];
    if (reward > 0) {
        rewards[_to] = 0;
```

```
        uint before = IBEP20(CAKE).balanceOf(address(this));
        _rewardsToken.withdraw(reward);
        uint cakeBalance = IBEP20(CAKE).balanceOf(address(this)).sub(before);

        IBEP20(CAKE).safeTransfer(msg.sender, cakeBalance);
        emit ProfitPaid(_to, cakeBalance, 0);
    }
}
```

## Issue #3: Attacker can drain funds by spamming relayer

| ID | Summary | Severity |
|---|---|---|
| THE-BUNNY-003 | An attacker can spam the relayer's withdraw API causing the relayer to burn ETH | High |

The relayer does not validate whether the user's account has any assets before submitting a transaction to the Ethereum blockchain. This allows an attacker to repeatedly call the relayer's withdraw API and cause the relayer to submit Ethereum transactions wasting ETH on gas fees.

The relayer API does not validate if the user's account is whitelisted, so this attack is possible even during the closed beta.

### Fix

PancakeBunny fixed the issue by validating the user's account contains assets with calls to VaultCollateral's collateralOf and VaultRelayer's balanceOf functions.

```
const vaultCollateral = new ethers.Contract(relayInternal.collateralContract,
VaultCollateralAbi, providers.eth)
const vaultRelayer = new ethers.Contract(VaultRelayer.address, VaultRelayer.abi,
providers.bsc)
const [collateral, balance, nonce] = await Promise.all([
    vaultCollateral.collateralOf(account),
    vaultRelayer.balanceOf(payload.pool, account),
    vaultRelayer.nonces(account)
])

if (collateral.isZero() && balance.isZero()) return false
```

An attacker could still spam withdraw requests, but they would need to add collateral each time and spend their own ETH on gas fees.

PancakeBunny may want to consider adding a transaction fee based on the estimated gas fees. This would discourage withdraws when Ethereum gas fees are high.

## Issue #4: Relayer does not properly check for chain reorg

| ID | Summary | Severity |
|---|---|---|
| THE-BUNNY-004 | Relayer does not properly check for a reorganization of the Ethereum blockchain during a deposit | High |

Any service that relies on information retrieved from a blockchain needs to ensure that the information will not change due to a reorganization. On the Ethereum blockchain, it is recommended that developers wait for 12 confirmations before relying on the information. While the relayer waits for 12 confirmations, it fails to properly check that the information has not changed during that timeframe.

At a high level, the relayer processes a deposit by:

1. User submits a deposit request to the relayer via HTTP API
2. Relayer validates the request and retrieves the amount of collateral the user deposited in VaultCollateral using collateralOf
3. Relayer waits for 12 confirmations from the current block number
4. Relayer again retrieves the amount of collateral the user deposited and checks that it is the same as the value retrieved in step 2
5. Relayer calls VaultRelayer's depositBySig

However, step 4 is not properly implemented and does not actually perform a check that the value is valid (e.g. a reorganization could happen that decreased the amount of collateral). This is because the call to collateralOf does not specify a blockTag with the block number from the call in step 2:

```
const collateralCurrent = await vaultCollateral.collateralOf(item.account)
if (collateralStashed.eq(ZERO) || collateralStashed.lt(collateralCurrent)) {
    await clearRelayItem(item.account)
    return
}
```

Additionally, the check in the if-statement is incorrect. It should return if collateralStashed is greater than collateralCurrent.

### Fix

PancakeBunny fixed the issue by specifying a blockTag when calling collateralOf in step 4, as well as changing the condition in the if-statement to check for equality.

```
const collateralCurrent = await vaultCollateral.collateralOf(item.account,
{ blockTag: item.blockETH })
if (collateralStashed.eq(ZERO) || !collateralStashed.eq(collateralCurrent)) {
    await clearRelayItem(item.account)
    return
}
```

## Issue #5: Relayer liqudation fails to check queue state

| ID | Summary | Severity |
|---|---|---|
| THE-BUNNY-005 | The relayer liquidator may overwrite an existing item in the queue causing a withdraw to be partially completed | Low |

The relayer architecture is based around a queue where each item is keyed by the account. As such, an account may only have one entry in the queue at a time. An attempt to submit a deposit or withdraw request when one is already pending in the queue will fail. This is essential to prevent race conditions between multiple requests or failures due to requests being interrupted.

However, the relayer liquidator will also add an entry to the queue for an account if the account needs to be liquidated and it failed to check for an existing entry. This may have caused a withdraw request to be only partially completed.

### Fix

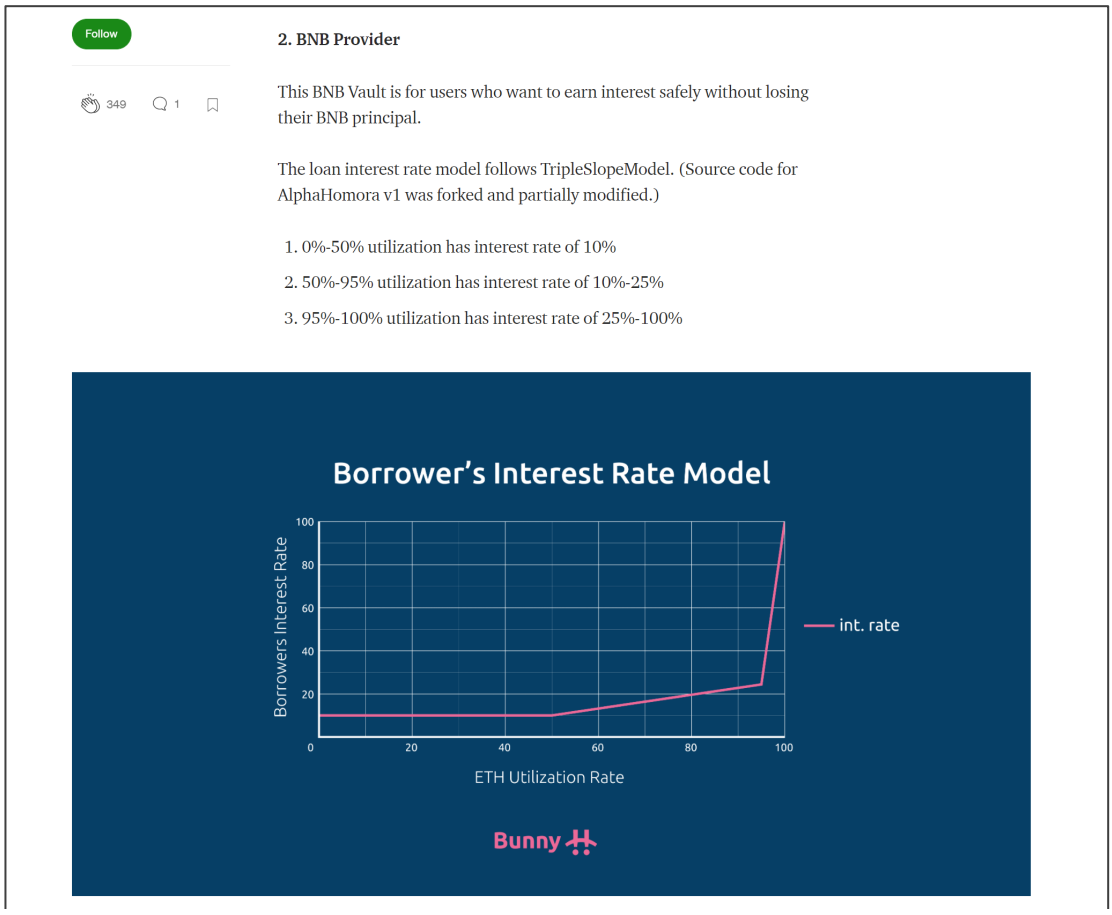
PancakeBunny fixed the issue by checking if an entry for the account is already in the queue. If so, the liquidation will be delayed.

One possible concern with this fix is that it requires the existing entry in the queue to make progress. While we did not find any issue that would cause an entry to stall in the queue forever, blockchain congestion could delay items in the queue. There is also the possibility of a user spamming requests to try to always have an item in the queue, e.g. to prevent liquidation, but the validation steps in the relayer should prevent this. Thus, the current fix should be adequate.

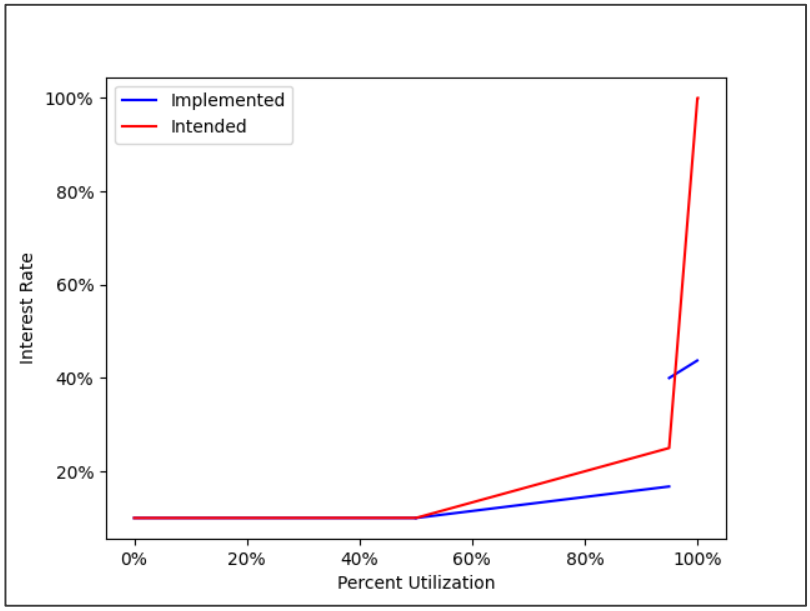
## Issue #6: Incorrect rate calculation in TripleSlopeModel

| ID | Summary | Severity |
|---|---|---|
| THE-BUNNY-006 | The interest rate calculated by TripleSlopeModel does not match the intended interest rate model | Low |

The ETH pool relayer does not immediately move funds from the Ethereum blockchain to the BSC blockchain. Instead, it borrows BNB from the PancakeBunny's BNB Venus pool. To compensate depositors, the ETH pool users pay interest on the borrowed assets. The interest rate model is documented as a triple slope model depending on utilization:




The code that implements the triple slope model does not match the documentation. It produces a significantly different outcome for utilizations above 50%:




This is due to incorrect numbers in the code:

```
uint utilization = debt.mul(10000).div(total);
if (utilization < 5000) {
    // Less than 50% utilization - 10% APY
    return uint(10e16) / 365 days;
} else if (utilization < 9500) {
    // Between 50% and 95% - 10%-25% APY
    return (10e16 + utilization.sub(5000).mul(15e16).div(10000)) / 365 days;
} else if (utilization < 10000) {
    // Between 95% and 100% - 25%-100% APY
    return (25e16 + utilization.sub(7500).mul(75e16).div(10000)) / 365 days;
} else {
    // Not possible, but just in case - 100% APY
    return uint(100e16) / 365 days;
}
```

## Fix

PancakeBunny fixed the issue by correcting the values in the code. The git commit is public at https://github.com/PancakeBunny-finance/Bunny/commit/bfa101e824dbc31a53c7cc6ed33a25d77c89ee49.

```
uint utilization = debt.mul(10000).div(total);
if (utilization < 5000) {
    // Less than 50% utilization - 10% APY
    return uint(10e16) / 365 days;
```

```
} else if (utilization < 9500) {
    // Between 50% and 95% - 10%-25% APY
    return (10e16 + utilization.sub(5000).mul(15e16).div(4500)) / 365 days;
} else if (utilization < 10000) {
    // Between 95% and 100% - 25%-100% APY
    return (25e16 + utilization.sub(9500).mul(75e16).div(500)) / 365 days;
} else {
    // Not possible, but just in case - 100% APY
    return uint(100e16) / 365 days;
}
```

## Issue #7: Total supply is modified before notifyDeposited

| ID | Summary | Severity |
|---|---|---|
| THE-BUNNY-007 | BunnyChef's updateRewards should be called before an update to a vault's total supply, but notifyDeposited is called after total supply is modified | Low |

BunnyChef allocates rewards to accounts based on their balance in each vault and each vault's allocation of rewards. The rewards for a vault are updated when needed: account rewards need to be updated, vault allocations change, or the token supply changes. The vault rewards are tracked as the number of reward tokens per share in the vault.

```
function notifyDeposited(address user, uint amount) external override onlyVaults
updateRewards(msg.sender) {
  ...
}

modifier updateRewards(address vault) {
    VaultInfo storage vaultInfo = vaults[vault];
    if (block.number > vaultInfo.lastRewardBlock) {
        uint tokenSupply = tokenSupplyOf(vault);
        if (tokenSupply > 0) {
            uint multiplier = timeMultiplier(vaultInfo.lastRewardBlock,
                block.number);
            uint rewards = multiplier.mul(bunnyPerBlock)
                 .mul(vaultInfo.allocPoint).div(totalAllocPoint);
            vaultInfo.accBunnyPerShare = vaultInfo.accBunnyPerShare
                 .add(rewards.mul(1e12).div(tokenSupply));
        }
        vaultInfo.lastRewardBlock = block.number;
    }
    _;
}
```

There are two times when the total supply of a vault changes: a deposit and a withdraw. BunnyChef has two functions that the vaults use for these events, notifyDeposited and notifyWithdrawn, respectively. Both functions will update the rewards for the vault before anything else. However, most vaults call notifyDeposited after already modifying their total supply, which may result in a lower reward amount.

For instance, consider VaultBunny's _deposit function:

```
function _deposit(uint amount, address _to) private nonReentrant notPaused {
    require(amount > 0, "VaultBunny: amount must be greater than zero");
    _totalSupply = _totalSupply.add(amount);
    _balances[_to] = _balances[_to].add(amount);
    _depositedAt[_to] = block.timestamp;
    _stakingToken.safeTransferFrom(msg.sender, address(this), amount);

    _bunnyChef.notifyDeposited(msg.sender, amount);
    emit Deposited(_to, amount);
}
```

_totalSupply is updated with the new amount before notifyDeposited is called, and thus, before BunnyChef's updateRewards can update the number of reward tokens per share.

### Fix

PancakeBunny fixed the issue by adding to each affected vault a call to BunnyChef's updateRewards before the total supply is updated. The git commit is public at https://github.com/PancakeBunny-finance/Bunny/commit/6f6bf82b5b436bd424d9e402d2e1631e840af2da.

```
function _deposit(uint amount, address _to) private nonReentrant notPaused {
    require(amount > 0, "VaultBunny: amount must be greater than zero");
    _bunnyChef.updateRewardsOf(address(this));
    _totalSupply = _totalSupply.add(amount);
    _balances[_to] = _balances[_to].add(amount);
    _depositedAt[_to] = block.timestamp;
    _stakingToken.safeTransferFrom(msg.sender, address(this), amount);

    _bunnyChef.notifyDeposited(msg.sender, amount);
    emit Deposited(_to, amount);
}
```

## Issue #8: Attacker can manipulate profit of Venus pool

| ID | Summary | Severity |
|---|---|---|
| THE-BUNNY-008 | Debt in Venus can be repaid by another account which can be used to increase the profit of a Venus pool that is borrowing | Medium |

A recent attack on PolygonBunny demonstrated that if an attacker can maliciously increase the profit of a vault, they will increase the performance fee which is multiplied by 1.20 (on PancakeBunny) and sent back to them as reward tokens. The Venus vaults may be susceptible to this attack because any account can repay the vaults' borrowings from Venus, which is used in the calculation of the vaults' balance.

```
function balance() public view override returns (uint) {
    uint debtOfBank = bank == address(0) ? 0 : IBank(bank).debtToProviders();
    return balanceAvailable().add(venusSupply).sub(venusBorrow).add(debtOfBank);
}

function updateVenusFactors() public {
    (venusBorrow, venusSupply) = safeVenus.venusBorrowAndSupply(address(this));
    ...
}
```

Currently, this attack is not possible because the Venus vaults are not borrowing from Venus. The vaults will only borrow when it is favorable based on the APY of borrowing vs lending as well as the value of XVS reward tokens. It is not trivial for an attacker to influence the decision to borrow because it is determined within _increaseCollateral which is only called by the keeper.

```
function harvest() public override accrueBank notPaused onlyKeeper {
    VENUS_BRIDGE_OWNER.harvestBehalf(address(this));
    _increaseCollateral(3);
}

function increaseCollateral() external onlyKeeper {
    _increaseCollateral(safeVenus.safeCompoundDepth(address(this)));
}
```

Since the attack is not currently practical, we did not thoroughly analyze whether an attack would be profitable. The attacker would need to have enough assets in the Venus vault to

earn a large enough performance fee to repay the borrowings. Also, the vault would need to have sufficient borrowings for the performance fee to be large.

### Fix

In PolygonBunny, this type of issue was fixed by keeping track of the balance internally instead of relying on values that could be manipulated by an attacker. A similar solution may be possible here as well.

As we discuss in the recommendations, it may be better to stop inflating the rate at which the performance fee is converted into reward tokens.

# Recommendations

These are recommendations to consider for improving PancakeBunny. They do not impose any immediate security impacts.

## Summary

| # | Title | Type | Importance |
|---|---|---|---|
| 1 | Limit amount of minted reward tokens | Safety | Major |
| 2 | Use oracle to determine minimum amounts for swaps | Safety | Minor |
| 3 | Price of LP tokens can be manipulated | Safety | Minor |

## Recommendation #1: Limit amount of minted reward tokens

The two recent attacks, against PancakeBunny and PolygonBunny, both involved minting of BUNNY reward tokens. The first attack was fixed by using an off-chain price oracle and the second attack was fixed by preventing the attacker from manipulating the performance fee. We recommend that additional mitigations are added to the minter to prevent exploitation of future issues.

One possible mitigation is to not mint reward tokens at all for the performance fee, and only buy tokens from the market. This would have prevented both attacks because the attacker would always be paying the current market price for each reward token. This would also eliminate the current inflation rate (FRE) on the performance fee, but inflation could still be achieved by rewarding tokens over time to stakers.

Another possible mitigation is to limit how many reward tokens will be minted during a block. While this may not completely prevent an attack, it would limit the impact on the market for the BUNNY reward tokens. In both recent attacks, the attacker needed to be able to mint a very large number of tokens for the attack to be profitable.

Lastly, reward tokens could be locked for some period of time to prevent flash loan attacks. If an attacker is unable to retrieve the reward tokens immediately, it is likely that the flash loan attack will fail due to inability to repay the flash loan. A well-resourced attacker might be able to exploit a vulnerability without a flash loan, but a time lock on the reward tokens would also allow PancakeBunny to monitor for abnormalities and potentially stop an attack.

## Recommendation #2: Use oracle to determine minimum amounts for swaps

A known threat for smart contracts that use distributed exchanges, such as PancakeBunny, is manipulating the price of token swaps. If an attacker can force a token swap to occur, for instance by calling a harvest function that gathers rewards and swaps them into another token, the attacker can manipulate the price at which the tokens are swapped. By doing so, the attacker may be able to earn a small profit.

PancakeBunny, and many similar smart contracts, avoid this issue by only allowing trusted accounts to call their harvest functions. This prevents an attacker from using a flash loan to manipulate prices, however, it does not stop an attacker who has sufficient assets. A well-resourced attacker might be able to sandwich the trusted harvest transaction between two of the attacker's transactions and earn a small profit by manipulating prices.

While this is a potential security risk, it is not mitigated in most DeFi smart contracts so for now we only recommend that it is fixed. Another mitigating factor is that the tokens on PancakeBunny with sufficiently large rewards to be interesting to an attacker are also highly liquid on PancakeSwap and hard to manipulate.

A solution to this problem would require using a trusted price oracle to determine a minimum swap amount. PancakeBunny could use the existing PriceCalcluator code, which uses ChainLink. The downside is an increase in gas usage for transactions that have a swap.

## Recommendation #3: Price of LP tokens can be manipulated

After a review of the PriceCalculator in PancakeBunny and considering the impact of the PolygonBunny attack, the only price manipulation left to an attacker is the price of liquidity provider (LP) tokens. The fair price of a LP token is a function of the price of each of its component tokens and how much of each component token a LP token may be redeemed for. The _getPairPrice in PriceCalculator gives the correct price for PancakeSwap tokens:

```
function _getPairPrice(address pair, uint amount) private view returns (uint
valueInBNB, uint valueInUSD) {
    address token0 = IPancakePair(pair).token0();
    address token1 = IPancakePair(pair).token1();
    uint totalSupply = IPancakePair(pair).totalSupply();
    (uint r0, uint r1,) = IPancakePair(pair).getReserves();

    uint sqrtK = HomoraMath.sqrt(r0.mul(r1)).fdiv(totalSupply);
    (uint px0,) = _oracleValueOf(token0, 1e18);
    (uint px1,) = _oracleValueOf(token1, 1e18);
    uint fairPriceInBNB = sqrtK.mul(2).mul(HomoraMath.sqrt(px0)).div(2 **
56).mul(HomoraMath.sqrt(px1)).div(2 ** 56);

    valueInBNB = fairPriceInBNB.mul(amount).div(1e18);
    valueInUSD = valueInBNB.mul(priceOfBNB()).div(1e18);
}
```

However, even though this function returns the correct price, that does not mean it cannot be manipulated by an attacker. For instance, an attacker could intentionally raise the price of a LP token by depositing only one component token into the LP token contract. It is also likely that an attacker would lose funds with the manipulation.

This is not inherently a security issue, but if PancakeBunny relies on the price of a LP token, it could become a security risk. For instance, in BunnyMinterV2's mintFor, the price of an asset, e.g. a LP token, is used to determine how many reward tokens to mint:

```
(uint contribution, ) = priceCalculator.valueOfAsset(asset, _performanceFee);
uint mintBunny = amountBunnyToMint(contribution);
```

We do not recommend any immediate solution, but only to remain aware of this risk. A possible mitigation for this security risk is to require that callers are not contracts, i.e., only allow EOA accounts. This prevents any price manipulation from flash loans. The downside is that some legitimate uses will be blocked and a loss of composability with other smart contracts.

# Conclusion

After a review of the PancakeBunny smart contracts, we found no major issues in its vault contracts nor its pot contracts. There were several issues with the relayer for the ETH pool, but this product is currently in closed beta which limits the impact. PancakeBunny quickly fixed the issues that we brought to their attention. We recommend that they reconsider how reward tokens are minted in exchange for the performance fee as this feature as led to two major attacks.