Pancake Swap

Security Assessment

August 8th, 2024 — Prepared by OtterSec

Nicholas R. Putra nicholas@osec.io

Robert Chen r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Pancake Swap engaged OtterSec to assess the `infinity-core` and `infinity-periphery` programs. This assessment was conducted between July 8th and August 29th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 7 findings throughout this audit engagement.

In particular, we identified a high-risk vulnerability concerning a rounding issue in the Bin update functionality, where the first LP may artificially inflate the value of shares by burning and donating tokens, resulting in significant losses for new LPs joining with different ratios (OS-PSP-ADV-00). Another issue involves transferring the native token's ERC20 representation and `msg.value` simultaneously, which may result in unintended balance updates, allowing an attacker to exploit the system by effectively doubling their balance (OS-PSP-ADV-01). Furthermore, we highlighted the possibility of a gas griefing attack (OS-PSP-ADV-02).

We also made recommendations for modifications to the codebase for improved efficiency (OS-PSP-SUG-02) and suggested modifying assembly bitmasking to ensure the correctness and safety of assembly operations (OS-PSP-SUG-01). Additionally, we advised incorporating certain missing validations to address potential security issues (OS-PSP-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/pancakeswap. This audit was performed against commit df64445.

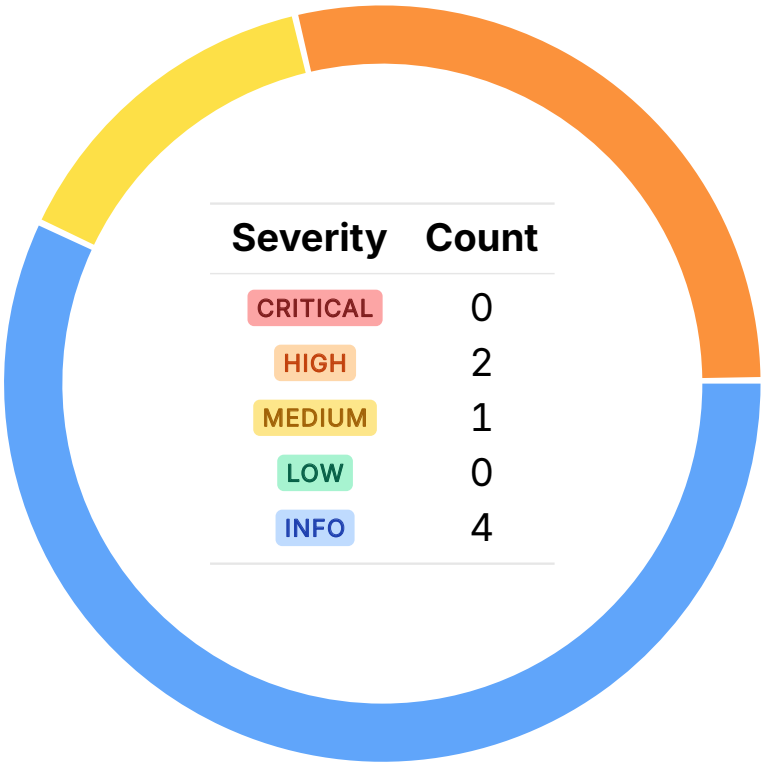
A brief description of the programs is as follows:

Name	Description
infinity-core	The core contracts of Pancake Swap, a multichain decentralized exchange utilizing an Automated Market Maker.
infinity-periphery	Periphery contracts which contain the logic that builds on top of the infinity-core logic.

03 — Findings

Overall, we reported 7 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	2
MEDIUM	1
LOW	0
INFO	4

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-PSP-ADV-00	HIGH	RESOLVED	Due to a rounding issue in `_updateBin`, the first LP may artificially inflate the value of shares by burning and donating tokens, resulting in significant losses for new LPs joining with different ratios.
OS-PSP-ADV-01	HIGH	RESOLVED	Transferring the native token's `ERC20` representation and `msg.value` simultaneously may result in unintended balance updates, allowing an attacker to exploit the system by effectively doubling their balance.
OS-PSP-ADV-02	MEDIUM	RESOLVED	In `_fetchProtocolFee`, the `call` may result in gas griefing by the assigned `protocolFeeController`.

Loss of Input Tokens Due to Incorrect Rounding HIGH OS-PSP-ADV-00

Description

There is a potential rounding issue in `BinPool::_updateBin` that may result in a significant loss for liquidity providers (LPs) under specific conditions (when there is only one LP in the active pool). In `_updateBin`, when liquidity is added to a bin, especially the active bin, a fee calculation mechanism is employed to ensure fair distribution among LPs and to collect protocol fees. A critical part of this process involves recalculating the shares minted and the composition fees based on the liquidity added and the fee structure. The vulnerability arises from how rounding errors are handled in these calculations.

>_ pancake-v4-core/src/pool-bin/libraries/BinPool.sol SOLIDITY

```
function _updateBin(State storage self, MintParams memory params, uint24 id, bytes32
    ↪  maxAmountsInToBin)
    internal
    returns (
        [...]
    )
{
        [...]
    if (id == activeId) {
        // Fees happens when user try to add liquidity in active bin but with different ratio of
            ↪  (x, y)
        /// eg. current bin is 40/60 (a,b) but user tries to add liquidity with 50/50 ratio
        uint24 lpFee = params.lpFeeOverride.isOverride()
            ? params.lpFeeOverride.removeOverrideAndValidate(LPFeeLibrary.TEN_PERCENT_FEE)
            : slot0Cache.lpFee;
        bytes32 feesAmount;
        (feesAmount, feeAmountToProtocol) =
            binReserves.getCompositionFeesAmount(slot0Cache.protocolFee, lpFee, amountsIn,
                ↪  supply, shares);
        compositionFeeAmount = feesAmount;
        if (feesAmount != 0) {
            {
                uint256 userLiquidity = amountsIn.sub(feesAmount).getLiquidity(price);
                /// @dev Ensure fee accrued only to existing lp, before calculating new share
                    ↪  for minter
                uint256 binLiquidity =
                    ↪  binReserves.add(feesAmount.sub(feeAmountToProtocol)).getLiquidity(price);
                shares = userLiquidity.mulDivRoundDown(supply, binLiquidity);
            }
            [...]
        [...]
    }[...]
}
```

The initial LP deposits liquidity into the pool. They burn their shares and donate a certain amount to inflate the value of each remaining share. When the new LP, unaware of the prior manipulation, adds

liquidity, due to the different ratio, composition fees are triggered. The calculation of `feesAmount` and `feeAmountToProtocol` is done in a way that may not perfectly align due to incorrect rounding. Since this calculation rounds down, the new LP will receive fewer shares than they would expect based on their input. Consequently, they will lose a majority of their input tokens due to the inflated value.

Remediation

Ensure to utilize proper rounding direction and prevent users to inflate the share values via `donate`.

Patch

Fixed in adf9b04.

Incorrect Token Synchronization HIGH OS-PSP-ADV-01

Description

In systems where a blockchain's native token also has an `ERC20` representation, such as `CELO` on the Celo network, this setup may result in an unintended exploit due to how balances are updated when both native and `ERC20` token transfers occur. When a user sends the native token as `msg.value` in a transaction, and the contract does not correctly handle the dual representation and syncing, the attacker may receive an additional settlement in the `ERC20` balance that they did not actually deposit. This effectively gives the attacker free tokens.

Remediation

Ensure careful handling of the synchronization between native token balances and their `ERC20` representations, such that any updates to one do not incorrectly affect the other.

Patch

Fixed in a096ade.

Possibility of Gas Griefing MEDIUM OS-PSP-ADV-02

Description

`ProtocolFees::_fetchProtocolFee` makes an external call to `protocolFeeController` to fetch the protocol fee. `call` is subject to a gas limit to ensure that it does not consume excessive gas. However, if `protocolFeeForPool` is implemented inefficiently and consumes a lot of gas, the external call may utilize the gas allotted to it, resulting in the calling contract (`ProtocolFees`) failing.

>_ *Breadcrumbspancake-v4-core/src/ProtocolFees.sol* SOLIDITY

```
function _fetchProtocolFee(PoolKey memory key) internal returns (bool success, uint24
    ↪  protocolFee) {
    if (address(protocolFeeController) != address(0)) {
        // note that EIP-150 mandates that calls requesting more than 63/64ths of remaining gas
        // will be allotted no more than this amount, so controllerGasLimit must be set with
            ↪  this
        // in mind.
        if (gasleft() < controllerGasLimit) revert ProtocolFeeCannotBeFetched();
        (bool _success, bytes memory _data) = address(protocolFeeController).call{gas:
            ↪  controllerGasLimit}(
            abi.encodeCall(IProtocolFeeController.protocolFeeForPool, (key))
        );
        [...]
    }
}
```

Remediation

Encapsulate the `call` in an inline assembly block to prevent the automatic copying of `returndata` to memory.

Patch

Fixed in d507e09.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-PSP-SUG-00	There are several instances where proper validation is not performed, resulting in potential security issues.
OS-PSP-SUG-01	Suggestion to modify bitmasking due to uncertainties around the Solidity compiler's handling of bit masking within assembly blocks, which may affect the correctness and safety of assembly operations.
OS-PSP-SUG-02	Recommendations for modifying the codebase for improved efficiency.
OS-PSP-SUG-03	Several gas optimizations may be incorporated into the codebase for enhanced efficiency and reduced gas costs.

Improvising Validation Logic OS-PSP-SUG-00

Description

1. Within `Vault::lock`, there is a potential `reentrancy` scenario. It is possible for the `locker` to execute a re-entrant call on `lock`, thus enabling the vault to become unlocked in the middle of the first `lock` process. Currently, there is no impact, but it would be appropriate to ensure that the `lock` is in an unlocked state before calling `lock`.

>_ pancake-v4-core/src/pool-bin/libraries/BinPool.sol SOLIDITY

```
function lock(bytes calldata data) external override returns (bytes memory result) {
    /// @dev only one locker at a time
    SettlementGuard.setLocker(msg.sender);
    result = ILockCallback(msg.sender).lockAcquired(data);
    /// @notice the caller can do anything in this callback as long as all deltas are
        ↪  offset after this
    if (SettlementGuard.getUnsettledDeltasCount() != 0) revert CurrencyNotSettled();
    /// @dev release the lock
    SettlementGuard.setLocker(address(0));
}
```

2. In `CLPool::Swap`, the condition that ensures the specified price limit for the swap is within acceptable boundaries (`sqrtPriceLimitX96 <= TickMath.MIN_SQRT_RATIO`) was updated to utilize a strict less than check (`<`) rather than a less than or equal to check (`<=`) in Uniswap. The same should be reflected in `CLPool::SwapP`.
3. In `CLPool::Swap`, modify the `swapFee` check (`if (state.swapFee == LPFeeLibrary.ONE_HUNDRED_PERCENT_FEE)`) to utilize `>=` instead of `==`.

>_ pool-cl/libraries/CLPool.sol SOLIDITY

```
function swap(State storage self, SwapParams memory params)
    internal
    returns (BalanceDelta balanceDelta, SwapState memory state)
{
    [...]
    if (state.swapFee == LPFeeLibrary.ONE_HUNDRED_PERCENT_FEE) {
        if (!exactInput) {
            revert InvalidFeeForExactOut();
        }
    }
    [...]
}
```

Remediation

Implement the above-mentioned suggestions.

Patch

1. Issue #1 was resolved in PR#118.
2. Issue #2 was resolved in 3c50a17.
3. Issue #3 was resolved in b68eab2.

Modifications to Assembly Bitmasking OS-PSP-SUG-01

Description

There are multiple modifications related to masking bits inside assembly blocks in Uniswap, as listed below, which should be incorporated into the current codebase:

1. In `VaultReserves::_getCurrencySlotKey`, `mstore(0x20, currency)` is changed to be masked by 20 bits.
2. In `LiquidityMath::addDelta`, `x` and `y` are masked as follows: `and(x, 0xffffffffffffffffffffffffffffffff), signextend(15, y)` in Uniswap.
3. In `Tick::update`, `liquidityGrossAfter` is masked to `div(sub(shl(128, 1), 1), numTicks)`, and in `tickSpacingToMaxLiquidityPerTick`, `tickSpacing` is masked to `signextend(2, tickSpacing)`.
4. In `ProtocolFeeLibrary::calculateSwapFee`, `self` is masked to `and(self, 0xfff)`.
5. `SqrtPriceMath` is changed to mask all of the values of `sqrtPX96` and `liquidity` on each `assembly` block. (`absDiff`) is also masked to `and(a, 0xffffffffffffffffffffffffffffffffffffffff)`), `and(b, 0xffffffffffffffffffffffffffffffffffffffff)`).
6. In `SwapMath::getSqrtPriceTarget`, `sqrtPriceNextX96` and `sqrtPriceLimitX96` are changed to be masked as well.
7. In `TickBitmap`, within `compress`, `flipTick`, `position`, and in `TickMath::getSqrtRatioAtTick`, `tick` and `tickSpaciing` are changed to be masked as `signextend(2, tick)` and `signextend(2, tickSpacing)`.
8. In `SwapMath::getSqrtPriceTarget`, mask `zeroForOne` (only mask the LSB, not one byte).

Remediation

Ensure to update the above-listed bitmaskings.

Patch

The above modifications were incorporated into the codebase.

Code Refactoring OS-PSP-SUG-02

Description

1. `MINIMUM_VALID_RESPONSE_LENGTH` should be updated from `0x40` (64 bytes) to `0xa0` (160 bytes) in `BinQuoter` and to `0xc0` (96 bytes) instead of `0x60` (192 bytes) in `CLQuoter`.
2. In `CLPositionManager`, for increased safety, it would be appropriate to move `_mint` to the end of the code.
3. `ProtocolFees::_fetchProtocolFee` does not need to return the `success` boolean as it is never utilized in the code.

Remediation

Incorporate the above modifications into the codebase.

Patch

1. Issue #1 was resolved in PR#14
2. Issue #3 was resolved in 1fba59f.

Gas Optimizations

OS-PSP-SUG-03

Description

1. Utilize the scratch space in `BinPosition` and `CLPosition` by packing data more efficiently through adjusting the order in which variables are written to memory.
2. In `CLPoolManager::modifyLiquidity` the check for a paused state may be optimized. In this case, calling `paused` may consume more gas than a simple comparison. The condition should be reordered to check `params.liquidityDelta > 0` first, as this is a cheaper comparison. Short-circuiting here will prevent the function call if liquidity modification is zero.

>_ src/pool-cl/CLPoolManager.sol SOLIDITY

```
function modifyLiquidity(
     PoolKey memory key,
     ICLPoolManager.ModifyLiquidityParams memory params,
     bytes calldata hookData
 ) external override returns (BalanceDelta delta, BalanceDelta feeDelta) {
     // Do not allow add liquidity when paused()
     if (paused() && params.liquidityDelta > 0) revert PoolPaused();
     [...]
}
```

3. In `Paussable`, replace the `bool` type with `uint256` to reduce execution gas costs. Boolean operations incur additional gas due to an extra `SLOAD` required to read, modify, and write back the slot's contents, whereas `uint256` avoids this overhead.
4. In `BinPositionManager`, implement a more efficient condition for reverting.

Remediation

Update the codebase with the above-stated optimizations.

Patch

Resolved in PR#180 and PR#30.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.