PancakeSwap Security Review

Pashov Audit Group

Conducted by: 0xbrivan, Hals, As3ros, crunter, Bretzel, solidit

April 28th 2025 - May 2nd 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **pancakeswap/cross-chain-swaps-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About PancakeSwap

The cross-chain contracts of PancakeSwap enable users to swap tokens between different blockchains by combining on-chain swaps with cross-chain bridging, supporting scenarios like swap→bridge, bridge→swap, and swap→bridge→swap. It consists of two main contracts — XChainSender (initiates swaps and bridges) and AcrossAdapter (handles cross-chain execution) — using a Dispatcher pattern for whitelisted function calls, currently integrating with Across Protocol for bridging.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - d8c1c31a8b92eaaac76280c543004904e882fef2

fixes review commit hash - cba10e3056e95f9bd6c9c8f19ea730b90e437945

Scope

The following smart contracts were in scope of the audit:

- `XChainSender`
- `Dispatcher`
- `AcrossAdapter`
- `ReentrancyGuardTransient`
- `Payments`
- `PCSOrder`
- `LibAddress`
- `Constants`
- `Commands`

7. Executive Summary

Over the course of the security review, 0xbrivan, Hals, As3ros, crunter, Bretzel, solidit engaged with PancakeSwap to review PancakeSwap. In this period of time a total of **8** issues were uncovered.

Protocol Summary

Protocol Name	PancakeSwap
Repository	https://github.com/pancakeswap/cross-chain-swaps-contracts
Date	April 28th 2025 - May 2nd 2025
Protocol Type	DEX

Findings Count

Severity	Amount
Medium	1
Low	7
Total Findings	**8**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Different token decimals in bridging lead to user losses	Medium	Resolved
[L-01]	Output amount incorrect when input and output tokens differ	Low	Resolved
[L-02]	Bridge relayerFeePct limits block small token transfers	Low	Resolved
[L-03]	Whitelist blacklist missing key ERC20 selector	Low	Resolved
[L-04]	Whitelisting bridge() enables bypassing single-bridge limit	Low	Resolved
[L-05]	Resetting approval to zero may fail with some non-standard tokens	Low	Resolved
[L-06]	Bridging fee-on-transfer tokens may fail transactions	Low	Acknowledged
[L-07]	Swaps with approve flow do not use full CONTRACT_BALANCE	Low	Resolved

8. Findings

8.1. Medium Findings

[M-01] Different token decimals in bridging lead to user losses

Severity

Impact: High

Likelihood: Low

Description

When bridging tokens via `AcrossAdapter::bridge`, the amount of output tokens that the relayer will send to the recipient on the destination chain is computed as follows:

```
uint256 outputAmt;
unchecked {
    // will not underflow as acrossData.relayerFeePct is less than 1e17 from
    // validation above
    outputAmt = (bridgeData.inputAmount *
      (1e18 - acrossData.relayerFeePct)) / 1e18;
}
```

This calculation does not account for the fact that some tokens, such as USDT, have different decimal configurations across chains. For instance, USDT uses 6 decimals on Arbitrum and 18 decimals on Binance chain. The impact depends on the user whether he cares about the minimum output amount and in both cases the impact is high:

```
if (outputAmt  < bridgeData.minOutputAmount) revert OutputAmountTooLow
  (bridgeData.minOutputAmount, outputAmt);
```

1. Bridging for example `60e6` USDT (representing $60 on Arbitrum) from Arbitrum to Binance should result in `60e18` in Binance, so the

`bridgeData.minOutputAmount` supplied by the user will be ~60e18. But the `outputAmt` does not scale decimals, so the value will always be much smaller and the above condition will always evaluate to false, causing the tx to always revert and leading to DoS.

2. If the user does not care about the slippage incurred in the output amount, the `minOutputAmt` could be set to zero, in which significant value loss will happen for the user. For example, sending `60e6` USDT (representing $60 on Arbitrum) would only result in `60e6` on Binance, which is worth approximately $0.000000000060.

Recommendations

Adjust the calculation to properly scale token amounts based on the decimal differences between source and destination chains.

8.2. Low Findings

[L-01] Output amount incorrect when input and output tokens differ

The `AcrossAdapter.bridge` function calculates the `outputAmt` based on the `inputAmount` and `relayerFeePct`, assuming that both input and output tokens have equivalent value and decimal precision. However, in cross-chain scenarios where inputToken and outputToken represent different assets (e.g., USDC → BTC), this calculation produces incorrect results.

For example in a real transaction on Base: https://basescan.org/tx/0x839fec7245fd4e9f64f94ae6f2d86fcced0a03a2361bdba04173077550c0f108

- inputToken: 0x0b2C639c533813f4Aa9D7837CAf62653d097Ff85 (USDC in OP mainnet).
- outputToken: 0xcbB7C0000aB88B473b1f5aFd9ef808440eed33Bf (cbBTC in Base).
- inputAmount: 6000000.
- outputAmount: 6071.

```
uint256 outputAmt;
        unchecked {
            // will not underflow as acrossData.relayerFeePct is less than 1e17
            // from validation above
            outputAmt = (bridgeData.inputAmount *
              (1e18 - acrossData.relayerFeePct)) / 1e18;
        }
```

It can lead to the bridge transaction not being filled.

[L-02] Bridge `relayerFeePct` limits block small token transfers

The `bridge` function in the `AcrossAdapter` contract enforces a strict range for the relayer fee percentage (`relayerFeePct`), limiting it between 0.001% (1e13)

and 10% (1e17):

```
// sanity check: relayer fee should not be gt 1e17 = 10% or lt 1e13 =
        // 0.001%
        if (acrossData.relayerFeePct > 1e17) revert RelayerFeePctTooHigh
          (acrossData.relayerFeePct);
        if (acrossData.relayerFeePct < 1e13) revert RelayerFeePctTooLow
          (acrossData.relayerFeePct);
```

These hard limits prevent the protocol from supporting small token transfers where fees would exceed the 10% maximum. When bridging small amounts, relayers may require higher percentage fees to compensate for fixed gas costs on the destination chain, especially when:

- The input amount is very minimal (e.g., 1-10 USDC).
- The destination chain has high gas costs.
- There are multiple commands to execute after bridging.

For example, bridging just 1 USDC from Optimism to Ethereum might require a 23% relayer fee to be economically viable for relayers, but the contract would reject this transaction due to the 10% cap.

[L-03] Whitelist blacklist missing key ERC20 selector

While the function correctly blacklists `transferFrom` (0x23b872dd) and `approve` (0x095ea7b3), it fails to blacklist other important ERC20 functions that could pose similar risks, such as `increaseAllowance` (0x39509351).

The `increaseAllowance` function, which is widely implemented in tokens using OpenZeppelin's ERC20 implementation (version 4.x.x and below), provides an alternative way to increase token allowances. If this selector is whitelisted for a token contract, it could potentially be exploited to approve the spending of tokens from the contract, circumventing the security measure intended by blacklisting approval.

```
function setSwapFunctionWhitelist
      (address target, bytes4 selector, bool status) external onlyOwner {
            if (target == address(0)) revert AddressZero();

            // blacklist 2 selector: 0x23b872dd: IERC20.transferFrom and
            // 0x095ea7b3: IERC20.approve
            if
              (selector == 0x23b872dd || selector == 0x095ea7b3) revert InvalidWhiteli

            swapFunctionWhitelisted[keccak256(abi.encode
              (target, selector))] = status;

            emit SetSwapFunctionWhitelist(target, selector, status);
        }
```

It's recommended to blacklist the selector `IERC20.increaseAllowance` (0x39509351).

[L-04] Whitelisting `bridge()` enables bypassing single-bridge limit

The `Dispatcher.setSwapFunctionWhitelist()` function is intended to restrict swap execution to safe, pre-approved functions on whitelisted target routers. At the same time, `XChainSender._validateCommands()` enforces that a user can only bridge once from the source chain by allowing a single `BRIDGE` command per order. However, this restriction can be bypassed if a malicious owner whitelists the selector for `AcrossAdapter.bridge()` as a swap function. A user can then craft a `Swap` command with `SwapData.input` targeting the `bridge()` function on the adapter. Since `SWAP` commands are not restricted in count, this allows a user to perform multiple bridging operations from the source chain by embedding them in `SWAP` commands, defeating the intended single-bridge restriction.

```
function setSwapFunctionWhitelist
      (address target, bytes4 selector, bool status) external onlyOwner {
        if (target == address(0)) revert AddressZero();

        // blacklist 2 selector: 0x23b872dd: IERC20.transferFrom and 0x095ea7b3:
        // IERC20.approve
        if
          (selector == 0x23b872dd || selector == 0x095ea7b3) revert InvalidWhitelistSe

        swapFunctionWhitelisted[keccak256(abi.encode
          (target, selector))] = status;

        emit SetSwapFunctionWhitelist(target, selector, status);
    }
```

Recommendation:
Prevent `bridge()` function selector (`0x2da334b5` for `bridge(bytes32,(bytes32,bytes32,uint256,uint256,address,address,bytes,uint256,bytes32,bytes))`) from being whitelisted as a swap function.

[L-05] Resetting approval to zero may fail with some non-standard tokens

The `Dispatcher.dispatch()` function, when executing a swap, approves the swap router on the token before swapping, and then resets the router's allowance back to zero after the swap. However, some tokens (such as BNB on BNB) revert when approving to zero, making these tokens incompatible with the protocol. As a result, swaps involving these tokens would fail, limiting token support and causing unexpected transaction reverts.

```
function dispatch(bytes32 orderId, PCSCommand[] memory pcsCommands) internal {
    //...
            if (!swapData.shouldTransferTokensBeforeSwap) {
                // safe practise: reset approval to 0 after swap
                ERC20(inputToken).safeApprove(swapData.target, 0);
            }
    //...
}
```

Same issue in `AcrossAdapter.bridge()` when `inputToken` allowance is reset to zero after bridging:

```
function bridge(bytes32 orderId, BridgeData memory bridgeData)
        external
        payable
        override
        nonReentrant
        whenNotPaused
{
    //...
    if (!isNativeInput) {
        // safe practise: reset allowance to 0
        ERC20(bridgeData.inputToken.toAddress()).safeApprove(address
          (v3spokePool), 0);
    }
    //...
}
```

Recommendation: Check the allowance of the target swap router if it has been fully consumed before resetting it back to zero.

[L-06] Bridging fee-on-transfer tokens may fail transactions

The protocol is designed to support all types of ERC20 tokens, including those with non-standard behaviors such as Fee-On-Transfer (FOT). However, FOT tokens are not properly handled during the bridging process, which can lead to transaction reverts.

```
function dispatch(bytes32 orderId, PCSCommand[] memory pcsCommands) internal {
    // --SNIP
    if (pcsCommand.command == Commands.SWAP) {
        // --SNIP
    } else if (pcsCommand.command == Commands.BRIDGE) {
        (BridgeData memory bridgeData) = abi.decode(pcsCommand.commandData,
          (BridgeData));
        address bridgeAdapter = bridgeData.target;
        // --SNIP
        if (inputToken == Constants.ETH) {
            // --SNIP
        }else {
            uint256 inputBalance = ERC20(inputToken).balanceOf(address(this));
==>             if (inputBalance < bridgeData.inputAmount) {
                revert InsufficientBridgeInputBalance();
            }
        }
    }
}
```

In the code above, `bridgeData.inputAmount` is set by the user based on the amount transferred to the contract when calling `XChainSender::send`. For FOT tokens, the actual amount received by the contract is lower due to the transfer fee, resulting in `inputBalance` being less than `bridgeData.inputAmount` and causing the transaction to revert.

An easy way to mitigate the problem is to allow users to specify `Constants.CONTRACT_BALANCE` as the bridge input amount, like the approach used in swap commands. This enables the contract to dynamically use its actual token balance, thus avoiding reverts due to FOT mechanics.

[L-07] Swaps with approve flow do not use full `CONTRACT_BALANCE`

The `dispatch` function in the Dispatcher contract allows specifying `Constants.CONTRACT_BALANCE` as the `inputAmount` for a `SWAP` command. This

is designed to signal that the swap should use the contract's entire balance of the specified `inputToken`, which is particularly useful when the exact balance is determined dynamically by previous operations (such as receiving funds from the Across bridge or from previous swaps).

The issue occurs specifically when using the approval flow (`shouldTransferTokensBeforeSwap = false`). While the contract correctly sets the approval amount to the full balance, it fails to update the calldata sent to the swap router:

```
function dispatch
      (bytes32 orderId, PCSCommand[] memory pcsCommands) internal {
        ...
                if (pcsCommand.command == Commands.SWAP) {
                    ...
                    uint256 inputAmount = swapData.inputAmount;
                    address inputToken = swapData.inputToken;

                    bool success;
                    if (inputToken != Constants.ETH) {
                        if (inputAmount == Constants.CONTRACT_BALANCE) {
                            inputAmount = ERC20(inputToken).balanceOf(address
                              (this));
                        }
                        // if inputToken is native, it will be sent with the
                        // transaction
                        if (swapData.shouldTransferTokensBeforeSwap) {
                            ERC20(inputToken).safeTransfer
                              (swapData.target, inputAmount);
                        } else {
                            ERC20(inputToken).safeApprove
                              (swapData.target, inputAmount);
                        }

                        (success,) = swapData.target.call
                        //(swapData.input); // @audit swapData.input is stale, still c

                        if (!swapData.shouldTransferTokensBeforeSwap) {
                            // safe practise: reset approval to 0 after swap
                            ERC20(inputToken).safeApprove(swapData.target, 0);
                        }
            ...
        }
    }
```

The issue is that while the contract updates the local `inputAmount` variable and sets the correct approval amount, `swapData.input` (the calldata payload for the swap router) is never updated to reflect the actual full balance. It still contains the original encoded parameters with the outdated amount.

For example, when calling PancakeSwap's SmartRouter with an `exactInputSingle` function:

```
struct ExactInputSingleParams {
    address tokenIn;
    address tokenOut;
    uint24 fee;
    address recipient;
    uint256 amountIn;
    uint256 amountOutMinimum;
    uint160 sqrtPriceLimitX96;
}
```

Swap routers executing the approve flow typically read the `amountIn` directly from their calldata (`swapData.input`) and then use `transferFrom` to pull that specific amount from the caller (Dispatcher). Because the `amountIn` in the calldata was not updated, the router will attempt to pull the original amount, not the full approved balance. This leads to the swap either failing or using only a portion of the intended funds.

Some routers like the PancakeSwap V3SwapRouter have special handling for CONTRACT_BALANCE, but this only works correctly in the direct transfer flow, not in the approval flow:

```
function exactInputSingle(ExactInputSingleParams memory params)
        external
        payable
        override
        nonReentrant
        returns (uint256 amountOut)
    {
        // use amountIn == Constants.CONTRACT_BALANCE as a flag to swap the
        // entire balance of the contract
        bool hasAlreadyPaid;
        if (params.amountIn == Constants.CONTRACT_BALANCE) {
            hasAlreadyPaid = true;
            params.amountIn = IERC20(params.tokenIn).balanceOf(address(this));
        }

        amountOut = exactInputInternal(
            params.amountIn,
            params.recipient,
            params.sqrtPriceLimitX96,
            SwapCallbackData({
                path: abi.encodePacked
                  (params.tokenIn, params.fee, params.tokenOut),
                payer: hasAlreadyPaid ? address(this) : msg.sender
            })
        );
        require(amountOut >= params.amountOutMinimum);
    }
```

It leads to:

- Swaps using the approval flow (`shouldTransferTokensBeforeSwap = false`) with `CONTRACT_BALANCE` will not use the full available balance
- Users receive fewer output tokens than expected.

Recommendations:

- Option 1: Disallow the combination of `inputAmount = Constants.CONTRACT_BALANCE` and `shouldTransferTokensBeforeSwap = false`.

```
if
      (swapData.inputAmount == Constants.CONTRACT_BALANCE && !swapData.shouldTransferT
        revert IncompatibleSwapFlags(); // Define this custom error
    }
```

- Option 2: Update the `amountIn` parameter within the `swapData.input`. However, it requires encode and decode the calldata.