hexens × PancakeSwap

SECURITY REVIEW REPORT FOR PANCAKESWAP

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

EXECUTIVE SUMMARY

OVERVIEW

This reports covers the results from multiple iterative audits of the periphery smart contracts of PancakeSwap Infinity.

Our security assessment was a full review of the code in multiple iterations for a combined total of 5 weeks.

During our audits, we have identified one critical severity vulnerability, which would have lead to insufficient slippage protection and could have allowed an attacker to steal assets from users.

We have also identified several minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

SCOPE

The analyzed resources are located on:

First Scope:
Issue Code: CAKE3-

https://github.com/pancakeswap/pancake-v4-periphery/commit/a5e23bb8852478d688f119ca42ad61b1bab2198a

Second Scope:
Issue Code: CAKE2-

https://github.com/pancakeswap/pancake-v4-periphery/tree/4d97f3a33a1aa5eabd7d1c4f843b528bf342cc5e

Third Scope:
Issue Code: CAKE4-

https://github.com/pancakeswap/infinity-periphery/tree/481650d4079e213b9d036ffedc908db9b18ffcf5

The issues described in this report were fixed in the following commits:

First Scope:
Issue Code: CAKE3-

https://github.com/pancakeswap/pancake-v4-periphery/commit/72633c4aab14b7775b85050a1c1932387ac1621e

Second Scope:
Issue Code: CAKE2-

https://github.com/pancakeswap/pancake-v4-periphery/commit/5a904ecdd6b445fc88d600bcf659209f534ad589

https://github.com/pancakeswap/pancake-v4-periphery/commit/7ca22462252ba93f86a4a1bd702c5548ea3cc330

https://github.com/pancakeswap/pancake-v4-periphery/commit/50e30cb7eb536d6f586d917fd061f9861c6d83c2

Third Scope:
Issue Code: CAKE4-

https://github.com/pancakeswap/infinity-periphery/tree/f27de0269032a247684486209b25b3eb8002004f

AUDITING DETAILS

STARTED	DELIVERED
CAKE3- 19.08.2024	04.09.2024
CAKE2- 21.10.2024	31.10.2024
CAKE4- 03.02.2025	11.02.2025

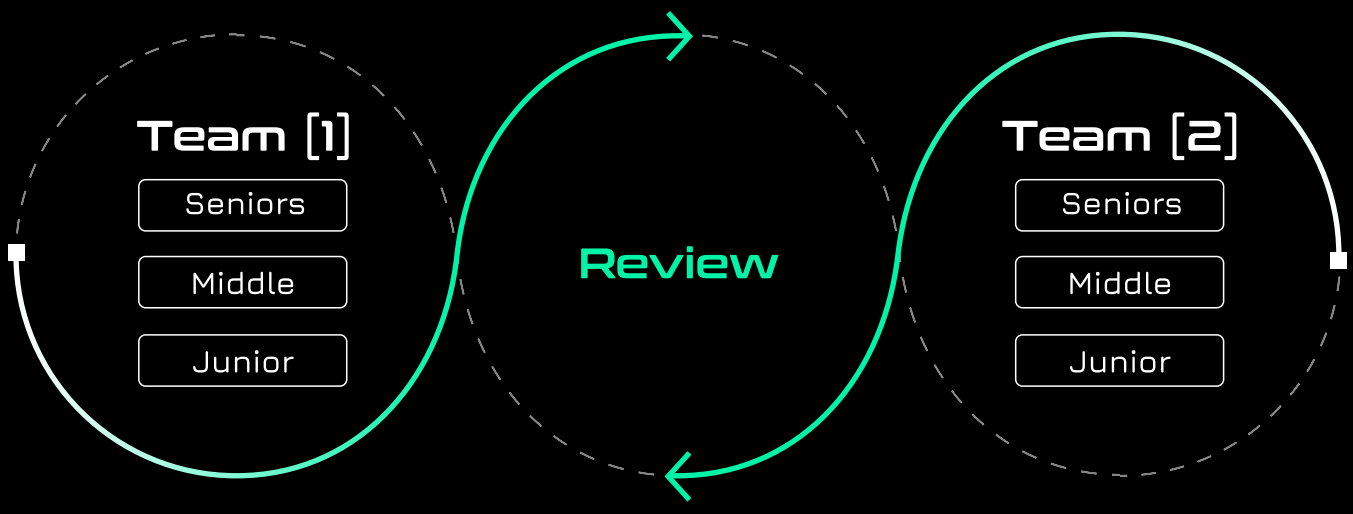
Review Led by

KASPER ZWIJSEN

Head of Audits

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

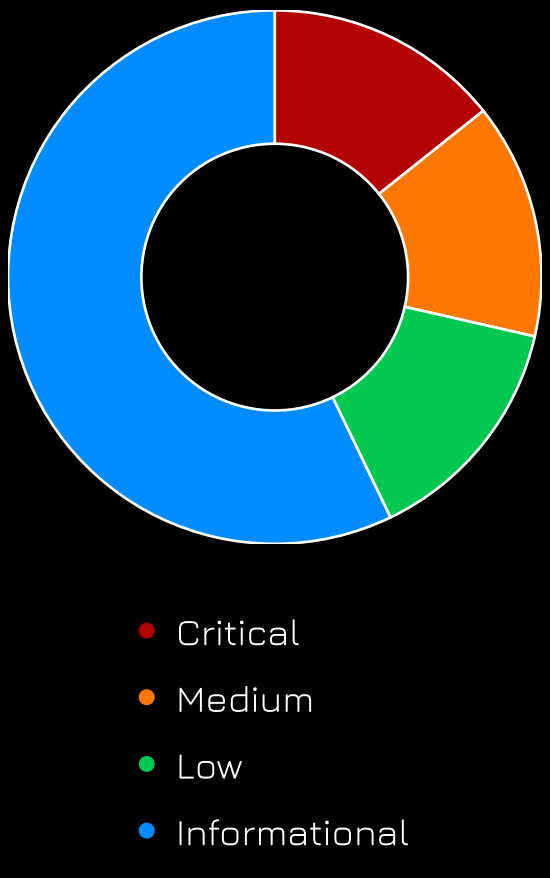
ISSUE SYMBOLIC CODES

Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	1
High	0
Medium	1
Low	1
Informational	4

Total: 7



WEAKNESSES

This section contains the list of discovered weaknesses.

CAKE3-13

BROKEN SLIPPAGE CHECK WHEN ADDING LIQUIDITY IN BINPOSITIONMANAGER

SEVERITY: Critical

PATH:

v4-periphery/src/pool-bin/BinPositionManager.sol#L189-L247

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The **_addLiquidity** function of the BinPositionManager should properly check for slippage when taking the user's token to increase liquidity.

The handle slippage, the function uses the **validateMaxIn** function, however it passes the incorrect parameter values for token 0 and token 1, and instead passes the value for token 0 twice:

```
delta.validateMaxIn(params.amount0Max, params.amount0Max);
```

This should be corrected to:

```
delta.validateMaxIn(params.amount0Max, params.amount1Max);
```

In the documentation string of the **validateMaxIn** function we also see that it expects both token 0 and token 1, as it uses these values to check the amount 0 and amount 1 from the delta.

The slippage check for token 1 is now incorrect and can result in direct asset loss through pool manipulation and sandwich attacks.

```
   function _addLiquidity(IBinPositionManager.BinAddLiquidityParams calldata
params) internal {


        -- SNIP --
         /// Slippage checks, similar to CL type. However, this is different
from TJ, in PCS v4,
          /// as hooks can impact delta (take extra token), user need to be
protected with amountMax instead
        delta.validateMaxIn(params.amount0Max, params.amount0Max);
}
```

```
/// @notice Revert if one or both deltas exceeds a maximum input
     /// @param delta The principal amount of tokens to be added, does not
include any fees accrued (which is possible on increase)
    /// @param amount0Max The maximum amount of token0 to spend
    /// @param amount1Max The maximum amount of token1 to spend
    /// @dev This should be called when adding liquidity (mint or increase)
     function validateMaxIn(BalanceDelta delta, uint128 amount0Max, uint128
amount1Max) internal pure {
        // Called on mint or increase, where we expect the returned delta to
be negative.
           // However, on pools where hooks can return deltas on modify
liquidity, it is possible for a returned delta to be positive (even after
discounting fees accrued).
        // Thus, we only cast the delta if it is guaranteed to be negative.
        // And we do NOT revert in the positive delta case. Since a positive
delta means the hook is crediting tokens to the user for minting/increasing
liquidity, we do not check slippage.
          // This means this contract will NOT support _positive_ slippage
checks (minAmountOut checks) on pools where the hook returns a positive
delta on mint/increase.
        int256 amount0 = delta.amount0();
        int256 amount1 = delta.amount1();
        if (amount0 < 0 && amount0Max < uint128(uint256(-amount0))) {
```

```
                revert MaximumAmountExceeded(amount0Max, uint128(uint256(-
amount0)));
        }
        if (amount1 < 0 && amount1Max < uint128(uint256(-amount1))) {
                revert MaximumAmountExceeded(amount1Max, uint128(uint256(-
amount1)));
        }
    }
```

It is recommended:

```
   function _addLiquidity(IBinPositionManager.BinAddLiquidityParams calldata
params) internal {

        -- SNIP --
         /// Slippage checks, similar to CL type. However, this is different
from TJ, in PCS v4,
          /// as hooks can impact delta (take extra token), user need to be
protected with amountMax instead
--         delta.validateMaxIn(params.amount0Max, params.amount0Max);
++         delta.validateMaxIn(params.amount0Max, params.amount1Max);

}
```

POSITIONS MAY BE TRANSFERRED & SUBSCRIPTIONS UPDATED WHILE VAULT/SETTLEMENTGUARD IS LOCKED

SEVERITY: Medium

PATH:

v4-periphery/pool-bin/BinFungibleToken.sol#L72-L75

v4-periphery/pool-cl/CLPositionManager.sol#L383

v4-periphery/pool-cl/base/CLNotifier.sol#L36-L39

v4-periphery/pool-cl/base/CLNotifier.sol#L58

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

Note: This finding has been included for completeness, as it was identified prior to the relevant fixes being merged during the audit window by the Pancake team independently.

CLPositionManager and **BinPositionManager** currently have no way of preventing callers from transferring positions regardless of **Vault's SettlementGuard** state, and positions may be transferred while the Vault is locked. In the case of CLPositionManager, there are similarly no restrictions on when a position may be subscribed or unsubscribed from.

This may result in unintended effects during **modifyLiquidities** calls, as it cannot be assumed that positions will maintain the same owner and subscription settings during Vault locks. It may also allow hooks to trigger notifications which may be inaccurate by the time after the liquidity has been modified. Malicious hooks may also re-enter Vault functions during

locked states before liquidity has been modified, by triggering the execution of the **notifySubscribe** callback in **CLNotifier::subscribe**.

```
    ...
        function batchTransferFrom(address from, address to, uint256[] calldata ids,
uint256[] calldata amounts)
        public
        virtual
        override
    ...
```

```
    ...
         function transferFrom(address from, address to, uint256 id) public
virtual override
    ...
```

```
    ...
          function subscribe(uint256 tokenId, address newSubscriber, bytes
calldata data)
        external
        payable
        onlyIfApproved(msg.sender, tokenId)
    ...
```

```
    ...
                function unsubscribe(uint256 tokenId) external payable
onlyIfApproved(msg.sender, tokenId) {
    ...
```

Consider implementing an onlyIfVaultUnlocked modifier to CLPositionManager and BinPositionManager, similar to the current Vault::isLocked modifier, which prevents execution if vault::getLocker() returns a nonzero address.

```
// to be available for both CLPositionManager and BinPositionManager
...
/// @notice reverts if vault is locked
modifier onlyIfVaultUnlocked() {
    if (vault.getLocker() != address(0)) revert VaultMustBeUnlocked();
    _;
}
...
```

```
// v4-periphery/interfaces/IPositionManager.sol

...
/// @notice Thrown when Vault must be unlocked
error VaultMustBeUnlocked();

...
```

CAKE2-1

FRONT RUNNING THE PERMIT FUNCTION CAN TRIGGER A DOS ATTACK

SEVERITY: Low

PATH:

src/base/SelfPermitERC721.sol#L15-L21

src/base/Permit2Forwarder.sol#L26-L31

src/base/Permit2Forwarder.sol#L17-L22

REMEDIATION:

Consider using the trustlessPermit() function, or ensure the contract does not revert when the permit() call fails.

STATUS: Fixed

DESCRIPTION:

The **Permit2Forwarder** and **SelfPermitERC721** contracts integrate the EIP-2612 Permit function, It transfers the burden of holding native (gas) tokens away from users, by allowing them to sign an approval off-chain and send it to a trusted service, which could use the funds as if the user called approve(). Because transactions are in the mempool and accessible to anyone, an attacker can extract the **_signature** parameters from the current call and front-run it with a direct **permit()** transaction. In this case, the result is harmful, as the user loses the functionality that follows the **permit()**.

For example, all functions that use the **permit** functionality for gasless transactions can be blocked, preventing users from using this feature within the project.

```
        function selfPermitERC721(address token, uint256 tokenId, uint256
deadline, uint8 v, bytes32 r, bytes32 s)
        public
        payable
        override
    {
         IERC721Permit(token).permit(address(this), tokenId, deadline, v, r,
s);
    }
```

```
    /// @notice allows forwarding batch permits to permit2
     /// @dev this function is payable to allow multicall with NATIVE based
actions
        function permitBatch(address owner, IAllowanceTransfer.PermitBatch
calldata _permitBatch, bytes calldata signature)
        external
        payable
    {
        permit2.permit(owner, _permitBatch, signature);
    }
```

```
    /// @notice allows forwarding a single permit to permit2
     /// @dev this function is payable to allow multicall with NATIVE based
actions
     function permit(address owner, IAllowanceTransfer.PermitSingle calldata
permitSingle, bytes calldata signature)
        external
        payable
    {
        permit2.permit(owner, permitSingle, signature);
    }
```

UNUSED IMPORTS

SEVERITY: Informational

PATH:

src/V4Router.sol::SafeCastTemp#L16

src/V4Router.sol::BalanceDelta#L7

src/V4Router.sol::PoolKey#L8

src/V4Router.sol::ActionConstants#17

src/pool-bin/BinMigrator.sol::SafeTransferLib#L5

src/pool-cl/CLMigrator.sol::SafeTransferLib#L5

src/pool-cl/CLMigrator.sol::Currency#L6

src/pool-cl/CLMigrator.sol::PoolKey#L8

REMEDIATION:

Remove unused imports.

STATUS: Fixed

DESCRIPTION:

The **V4Router.sol** contract imports **./libraries/SafeCast.sol**, but does not use it anywhere, moreover, the **SafeCastTemp** is already imported in the **BinRouterBase.sol** and **CLRouterBase.sol** contracts, from which the **V4Router.sol** inherits from.

There are also a couple of not-used imports, which are mentioned in the path.

```
import {SafeCastTemp} from "./libraries/SafeCast.sol";
```

```
import {BalanceDelta} from "pancake-v4-core/src/types/BalanceDelta.sol";
import {PoolKey} from "pancake-v4-core/src/types/PoolKey.sol";
```

```
import {ActionConstants} from "./libraries/ActionConstants.sol";
```

```
import {SafeTransferLib} from "solmate/src/utils/SafeTransferLib.sol";
```

```
import {SafeTransferLib} from "solmate/src/utils/SafeTransferLib.sol";
import {Currency} from "pancake-v4-core/src/types/Currency.sol";
```

```
import {PoolKey} from "pancake-v4-core/src/types/PoolKey.sol";
```

REDUNDANT ASSIGNMENT LIBRARY TO TYPE

SEVERITY: Informational

PATH:

src/types/Currency.sol#L10

REMEDIATION:

Remove redundant assignments of CurrencyLibrary wherever they occur.

STATUS: Fixed

DESCRIPTION:

The **CurrencyLibrary** library is assigned to the **Currency** type in multiple contracts, but in the **Currency.sol** contract, this library is already assigned globally to the **Currency** type.

```
using CurrencyLibrary for Currency global;
```

CUSTOM ERROR UPON REVERT INSIDE THE FUNCTION BINPOSITIONMANAGER::_ADDLIQUIDITY() MAY NOT BE IDEAL

SEVERITY: Informational

PATH:

v4-periphery/pool-bin/BinPositionManager.sol#L195-L197

REMEDIATION:

Consider adjusting the custom error so that it signifies that the revert happened because the activeId is not within slippage inside BinPositionManager::_addLiquidity():

if (params.activeIdDesired + params.idSlippage < activeId) revert IdSlippageCaught(params.activeIdDesired, params.idSlippage, activeId);

if (params.activeIdDesired - params.idSlippage > activeId) revert IdSlippageCaught(params.activeIdDesired, params.idSlippage, activeId);

STATUS: Fixed

DESCRIPTION:

BinPositionManager::_addLiquidity() reverts with the custom error **IdDesiredOverflows** (line 196-197 in v4-periphery/pool-bin/BinPositionManager.sol).
A better custom error would be as in TraderJoe V2's function **_addLiquidity()** to revert with something like **IdSlippageCaught**. This may be more correct to signify that this revert happened due to the id slippage was caught (activeId is not within slippage).

Also the custom error may be better if it includes the **params.idSlippage** and **activeId** similar to TraderJoe's implementation.

```
(uint24 activeId,,) = binPoolManager.getSlot0(params.poolKey.toId());
if (params.activeIdDesired + params.idSlippage < activeId) revert
IdDesiredOverflows(activeId);
if (params.activeIdDesired - params.idSlippage > activeId) revert
IdDesiredOverflows(activeId);
```

TYPOGRAPHICAL ERRORS

SEVERITY: Informational

REMEDIATION:

Consider fixing all typos before deployment.

STATUS: Fixed

DESCRIPTION:

1. infinity-periphery/src/MixedQuoter.sol

- In the function name **convertWETHToV4NativeCurency();** the word "**Currency**" is misspelled as "**Curency**".

```
function convertWETHToV4NativeCurency(PoolKey memory poolKey, address
tokenIn, address tokenOut)
```

2. infinity-periphery/src/pool-bin/interfaces/IBinPositionManager.sol

- In the error message **AddLiquidityInputActiveIdMismath();**, the word "**Mismatch**" is misspelled as "**Mismath**".

```
    error AddLiquidityInputActiveIdMismath();
```

- In the comment above **BinRemoveLiquidityParams** struct; the word "**receive**" is misspelled as "**recieve**".

```
    /// @notice BinRemoveLiquidityParams
    /// - amount0Min: Min amount to recieve for token0
    /// - amount1Min: Min amount to recieve for token1
```

3. infinity-periphery/src/Planner.sol
   - The comment below **finalizeSwap()**; the word "**isn't**" is misspelled as "**isnt**".

```
blindly settling and taking all, without slippage checks, isnt recommended
in prod
```

4. infinity-periphery/src/base/BaseActionsRouter.sol
   - In the comment above **msgSender()** function; the word "**shouldn't**" is misspelled as "**shouldnt**".

```
`msg.sender` shouldnt be used, as this will be the v4 vault contract that
calls `lockAcquired`
```

5. infinity-periphery/src/pool-cl/CLMigrator.sol
   - In the comment under the **migrateFromV2()** and **migrateFromV3()** functions; the word "**manually**" is misspelled as "**mannually**".

```
/// @notice if user mannually specify the price range, they might need to
send extra token
```

hexens × PancakeSwap