hexens × PancakeSwap

FEB.25

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 audit covered updates to the smart contracts of PancakeSwap V4 core and periphery. It included changes to the protocol fee implementation as well as renaming of V4 to Infinity.

Our security assessment was a full review changelog of the code, spanning a total of 1 week.

During our audit, we did not identify any security vulnerabilities. We did identify several code optimisations and informational issues.

Finally, 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:

- Core: https://github.com/pancakeswap/infinity-core/tree/8c8c34ad777ed7a3fb64c5b91a647e6f225a19da
- Periphery: https://github.com/pancakeswap/infinity-periphery/tree/481650d4079e213b9d036ffedc908db9b18ffcf5

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

- Core: https://github.com/pancakeswap/infinity-core/tree/a209a50fa2a87217042a5040b61a4d9e215f99f6
- Periphery: https://github.com/pancakeswap/infinity-periphery/tree/f27de0269032a247684486209b25b3eb8002004f
- Developer docs: https://github.com/pancakeswap/pancake-developer/tree/facf2553d9f4ab832e49ff2aab67f232060c67fa

AUDITING DETAILS

STARTED
03.02.2025

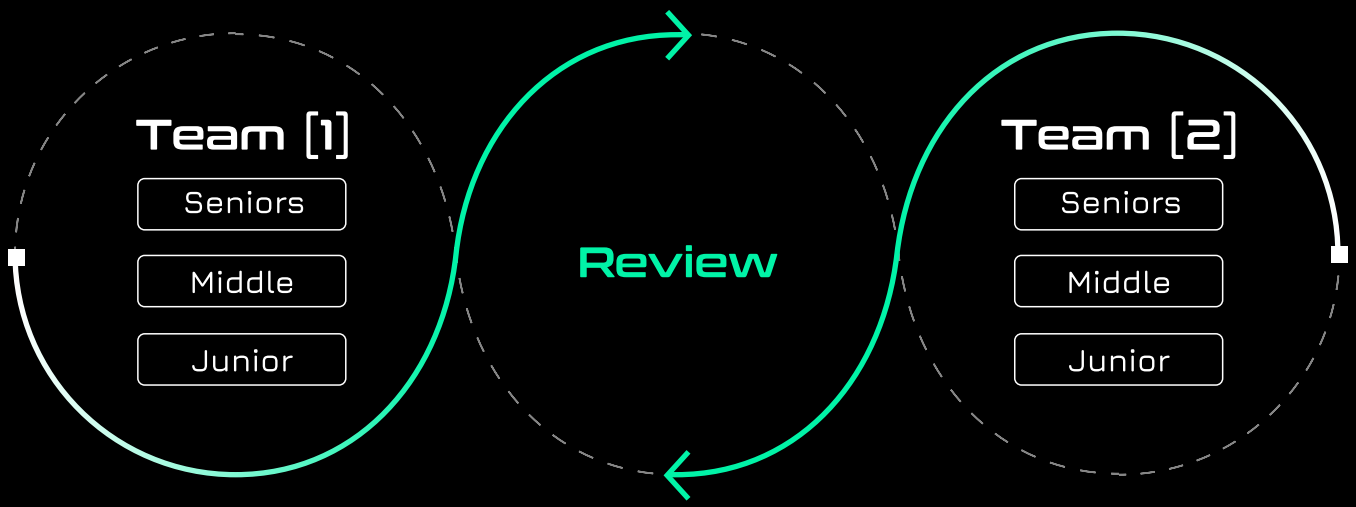
DELIVERED
11.02.2025

Review Led by

KASPER ZWIJSEN
Head of Audits | Hexens

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.

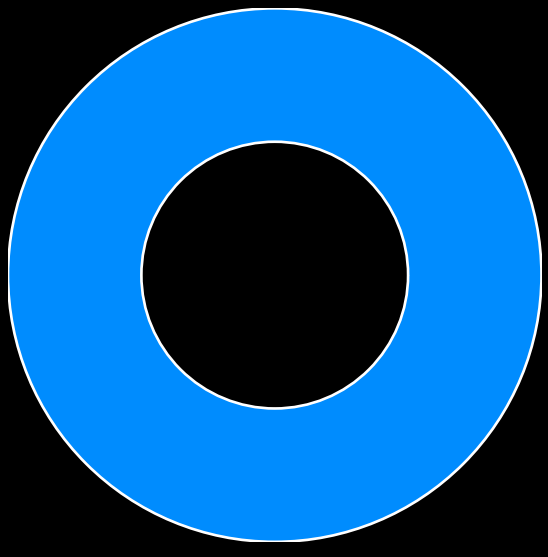
ISSUE SYMBOLIC CODES

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.

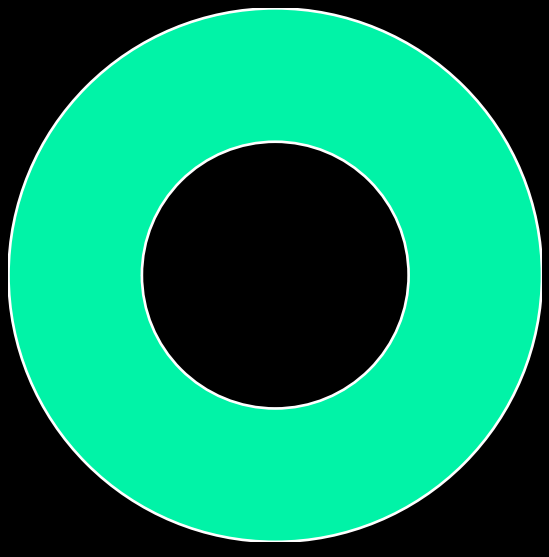
FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	0
Medium	0
Low	0
Informational	5

Total: 5



Informational



Fixed

WEAKNESSES

This section contains the list of discovered weaknesses.

CAKE4-1

TYPOGRAPHICAL ERRORS

SEVERITY: Informational

REMEDIATION:

Consider fixing all typos before deployment.

STATUS: Fixed

DESCRIPTION:

1. infinity-core/src/ProtocolFeeController.sol

- In the error message InvalidProtocolFeeSplitRatio();, the word "Invalid" is misspelled as “Invliad".

```
error InvliadProtocolFeeSplitRatio();
```

- In the natspec comment to the setProtocolFee() function, the word “protocol” is misspelled as “protcool”.

```
/// @notice Override the default protcool fee for the pool
```

2. 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)
```

3. 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
```

4. 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
```

5. 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`
```

6. 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
```

7. infinity-core/src/pool-cl/libraries/CLPool.sol
   - In the comment above the **PoolAlreadyInitialized()** error; the word "**initialize**" is misspelled as "**initalize**".

```
/// @notice Thrown when trying to initalize an already initialized pool
```

CONSTANT VARIABLES SHOULD BE MARKED AS PRIVATE

SEVERITY: Informational

PATH:

core/src/ProtocolFeeController.sol#L25

REMEDIATION:

The mentioned variables should be marked as private instead of public.

STATUS: Fixed

DESCRIPTION:

In the following locations, there are constant variables that are declared **public**. However, setting these constants to **private** will save deployment gas. This is because the compiler won't have to create non-payable getter functions for deployment calldata, won't need to store the bytes of the value outside of where it's used, and won't add another entry to the method ID table. If necessary, the values can still be read from the verified contract source code:

```
uint256 public constant ONE_HUNDRED_PERCENT_RATIO = 1e6;
```

CALL TO PROTOCOL FEE CONTROLLER COULD BE STATIC

SEVERITY: Informational

PATH:

ProtocolFees.sol:_fetchProtocolFee#L46-L78

REMEDIATION:

The low-level call should be done using staticcall to adhere to the interface and to ensure no state-changes can be made in custom implementations of the protocol fee controller.

STATUS: Fixed

DESCRIPTION:

The abstract contract **ProtocolFees** uses the **ProtocolFeeController** contract to calculate the fees for a given pool key. It does this using an external call.

Even though protocol fee controller address is configurable (although it is assumed to be **ProtocolFeeController.sol** in the repository), the function to-be-called is hardcoded to be **IProtocolFeeController.protocolFeeForPool**, which is a view function in both the interface and the contract.

However, a low-level **call** is used in assembly instead of **staticcall**.

```
    function _fetchProtocolFee(PoolKey memory key) internal returns (uint24
protocolFee) {
        if (address(protocolFeeController) != address(0)) {
            address targetProtocolFeeController = address(protocolFeeController);
            bytes memory data =
abi.encodeCall(IProtocolFeeController.protocolFeeForPool, (key));

            bool success;
            uint256 returnData;
            assembly ("memory-safe") {
                // only load the first 32 bytes of the return data to prevent gas
griefing
                success := call(gas(), targetProtocolFeeController, 0, add(data,
0x20), mload(data), 0, 32)

                // load the return data
                returnData := mload(0)

                // success if return data size is also 32 bytes
                success := and(success, eq(returndatasize(), 32))
            }

            // revert if call fails or return size is not 32 bytes
            if (!success) {
                CustomRevert.bubbleUpAndRevertWith(
                    targetProtocolFeeController, bytes4(data),
ProtocolFeeCannotBeFetched.selector
                );
            }

            if (returnData == uint24(returnData) && uint24(returnData).validate()) {
                protocolFee = uint24(returnData);
            } else {
                // revert if return value overflow a uint24 or greater than max
protocol fee
                revert ProtocolFeeTooLarge(uint24(returnData));
            }
        }
    }
```

INACCURATE EVENT DATA WHEN COLLECTING FEE-ON-TRANSFER TOKENS

SEVERITY: Informational

PATH:

src/ProtocolFeeController.sol

REMEDIATION:

Consider returning the difference in the balances before and after the transfer to get the exact fee amount.

STATUS: Fixed

DESCRIPTION:

The **collectProtocolFee** function is designed to collect protocol fees from the pool manager and can only be called by the contract owner:

```
    function collectProtocolFee(address recipient, Currency currency, uint256 amount)
external onlyOwner {
        IProtocolFees(poolManager).collectProtocolFees(recipient, currency, amount);

        emit ProtocolFeeCollected(currency, amount);
    }
```

The function calls **collectProtocolFees()** in the pool manager contract, which in turn transfers tokens from the **Vault** and returns the **amountCollected**:

```
    function collectProtocolFees(address recipient, Currency currency, uint256
amount)
        external
        override
        returns (uint256 amountCollected)
    {
        if (msg.sender != address(protocolFeeController)) revert InvalidCaller();

        amountCollected = (amount == 0) ? protocolFeesAccrued[currency] : amount;
        protocolFeesAccrued[currency] -= amountCollected;
        vault.collectFee(currency, amountCollected, recipient);
    }
```

```
    function collectFee(Currency currency, uint256 amount, address recipient)
external onlyRegisteredApp {
        // prevent transfer between the sync and settle balanceOfs (native settle
uses msg.value)
        (Currency syncedCurrency,) = VaultReserve.getVaultReserve();
        if (!currency.isNative() && syncedCurrency == currency) revert
FeeCurrencySynced();
        reservesOfApp[msg.sender][currency] -= amount;
        currency.transfer(recipient, amount);
    }
```

As a result, the **ProtocolFeeCollected** event may display an amount larger than what was truly collected, leading to inaccurate event data.

DISCREPANCY BETWEEN DOCUMENTATION AND PROTOCOLFEES IMPLEMENTATION

SEVERITY: Informational

PATH:

src/ProtocolFeeController.sol:setProtocolFee#L115-L120

REMEDIATION:

Consider updating the official documentation to align with the actual implementation.

STATUS: Fixed

DESCRIPTION:

According to natspec documentation and the source, the maximum value for **newProtocolFee** is 4000 which is equivalent to 0,4%.

```
    /// @param newProtocolFee 1000 = 0.1%, and max at 4000 = 0.4%. If set at
0.1%, this means 0.1% of amountIn for each swap will go to protocol
    function setProtocolFee(PoolKey memory key, uint24 newProtocolFee)
external onlyOwner {
        if (address(key.poolManager) != poolManager) revert
InvalidPoolManager();

        // no need to validate the protocol fee as it will be done in the
pool manager
        IProtocolFees(address(key.poolManager)).setProtocolFee(key,
newProtocolFee);
    }
```

```
    function setProtocolFee(PoolKey memory key, uint24 newProtocolFee) external
virtual {
        if (msg.sender != address(protocolFeeController)) revert InvalidCaller();
        if (!newProtocolFee.validate()) revert ProtocolFeeTooLarge(newProtocolFee);
        PoolId id = key.toId();
        _setProtocolFee(id, newProtocolFee);
        emit ProtocolFeeUpdated(id, newProtocolFee);
    }
```

```
uint24 internal constant FEE_0_THRESHOLD = 4001;
uint24 internal constant FEE_1_THRESHOLD = 4001 << 12;
```

```
function validate(uint24 self) internal pure returns (bool valid) {
        // Equivalent to: getZeroForOneFee(self) <= MAX_PROTOCOL_FEE &&
getOneForZeroFee(self) <= MAX_PROTOCOL_FEE
        assembly ("memory-safe") {
            let isZeroForOneFeeOk := lt(and(self, 0xfff), FEE_0_THRESHOLD)
            let isOneForZeroFeeOk := lt(and(self, 0xfff000), FEE_1_THRESHOLD)
            valid := and(isZeroForOneFeeOk, isOneForZeroFeeOk)
        }
    }
```

However, in documentation, the maximum protocol fee is **1000 (0.1%)** on token0 or token1, depending on swap direction. This discrepancy leads to an inconsistency between the documentation and the source code.


hexens
×
PancakeSwap