hashlock.

# Security Audit

## Honeypot Finance (DeFi)

August 2024 info@hashlock.com.au Hashlock Pty Ltd

# Table of Contents

## CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE THAT COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR THE USE OF THE CLIENT.

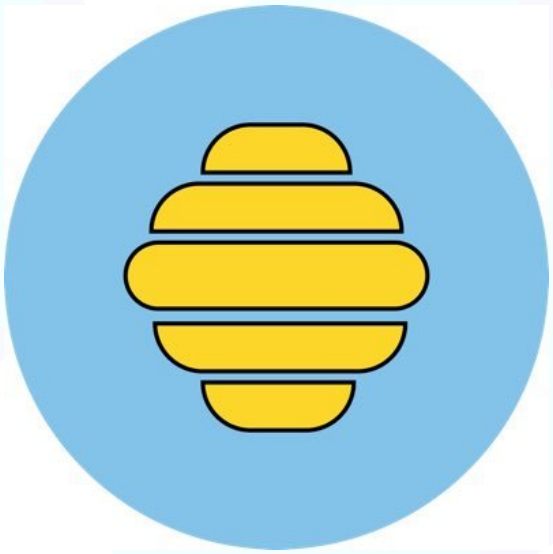
# Executive Summary

The Honeypot Finance team partnered with Hashlock to conduct a security audit of their YexFTOFacade.sol, YexFTOFactory.sol, YexFTOLaunchToken.sol, YexFTOPair.sol, BurnableHook.sol, NormalHook.sol, and VestingHook.sol smart contracts. Hashlock manually and proactively reviewed the code to ensure the project's team and community that the deployed contracts were secure.

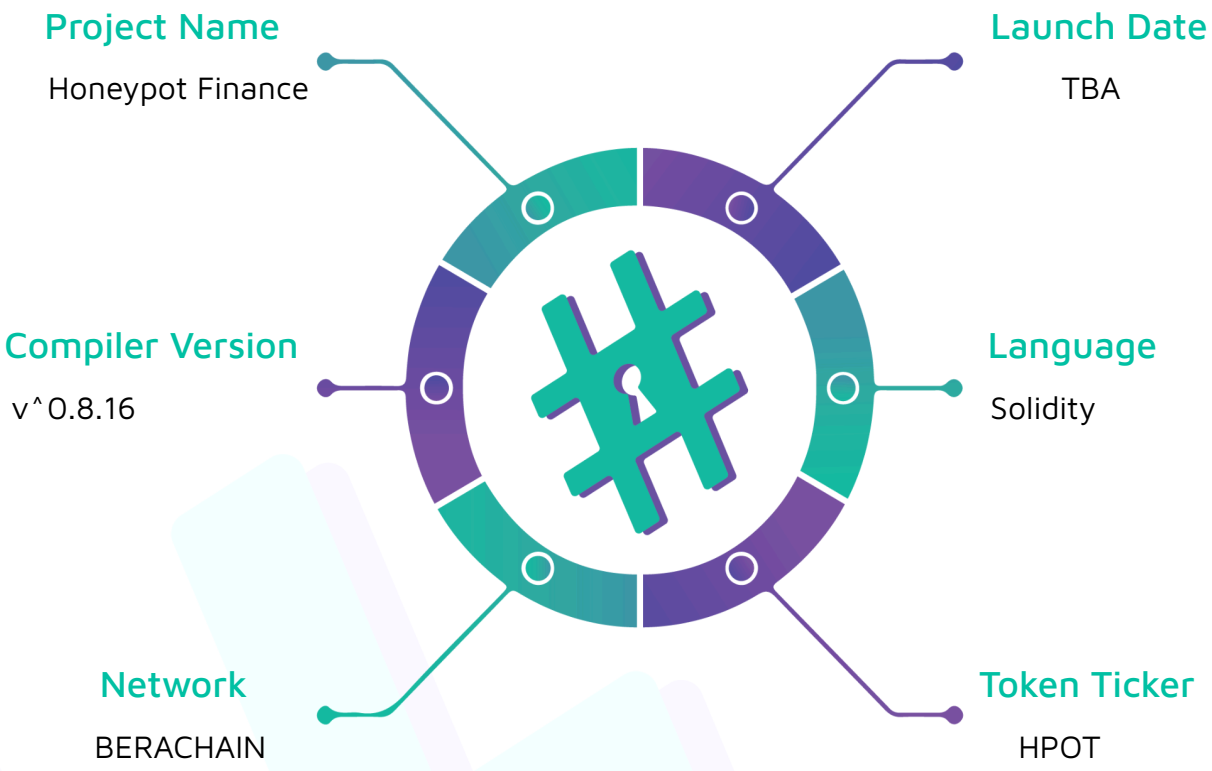
# Project Context

Honeypot Finance is building a community-run DeFi Hub on Berachain that integrates a unique AMM model to unite a community-led launchpad and DEX. It addresses the liquidity segmentation across DeFi products which leads to low liquidity utilization.

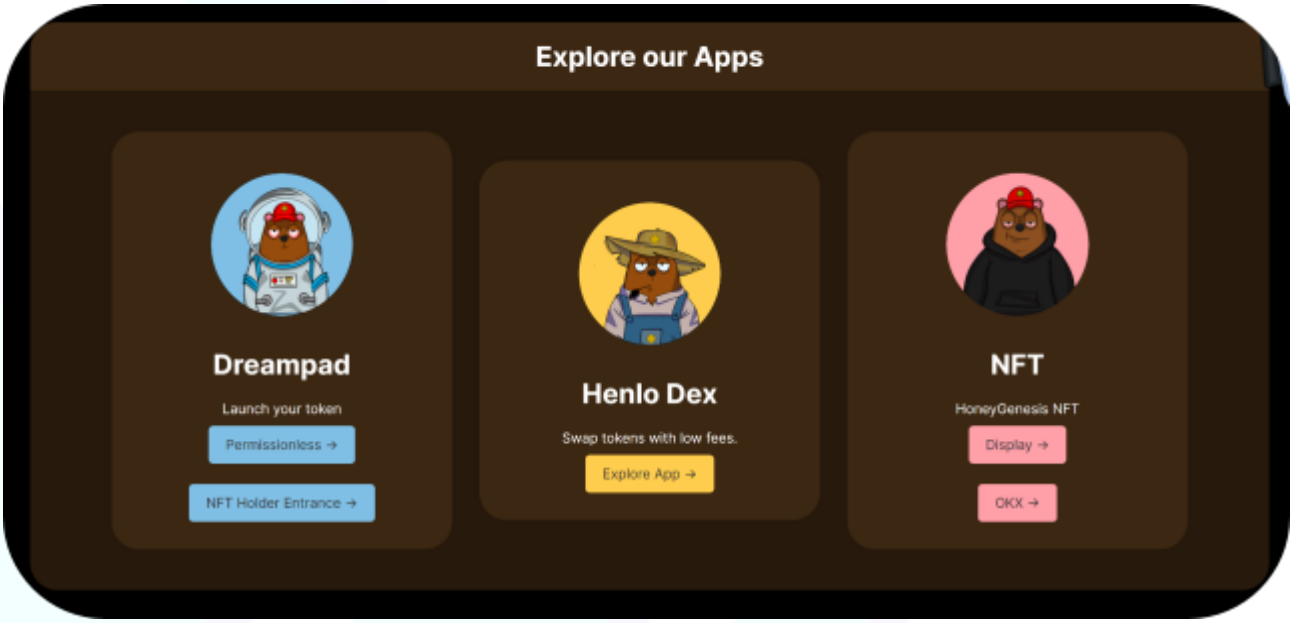
**Project Name**: Honeypot Finance
**Compiler Version:** ^0.8.16
**Website:** https://honeypotfinance.xyz/
**Logo:**



**Visualised Context:**



**Project Visuals:**

# Audit scope

We at Hashlock audited the solidity code within the Honeypot Finance project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

| Description | Honeypot Finance Protocol Smart Contracts |
|---|---|
| Platform | Berachain / Solidity |
| Audit Date | July, 2024 |
| Contract 1 | YexFTOFacade.sol |
| Contract 1 MD5 Hash | f55bc447022cc7e93b87d4ecd9332b04 |
| Contract 2 | YexFTOFactory.sol |
| Contract 2 MD5 Hash | f9ac83736e1216d0d1aa4c452f171658 |
| Contract 3 | YexFTOLaunchToken.sol |
| Contract 3 MD5 Hash | cae5c7c528a33aa3fe1442886ad0dfc4 |
| Contract 4 | YexFTOPair.sol |
| Contract 4 MD5 Hash | d794ba52c7dc32ffc095a1388be428a5 |
| Contract 5 | BurnableHook.sol |
| Contract 5 MD5 Hash | 05cbde8100f5ab708adf1ec0dba1a9c6 |
| Contract 6 | NormalHook.sol |
| Contract 6 MD5 Hash | ad277e698bd7f32a91cf98eb97393a8a |
| Contract 7 | VestingHook.sol |
| Contract 7 MD5 Hash | acddbacb4e3992de16622fffa2e6885c |

# Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts. We initially identified some significant vulnerabilities that have since been addressed.


*The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on-chain monitoring technology.*

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section.

All vulnerabilities initially identified have now been resolved and acknowledged.

**Hashlock found:**

1 High-severity vulnerabilities

2 Medium-severity vulnerabilities

4 Low-severity vulnerabilities

9 Gas Optimisations

**Caution:** *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

# Intended Smart Contract Behaviours

| Claimed Behaviour | Actual Behaviour |
|---|---|
| **YexFTOFacade.sol**<br>- Allows users to:<br>  - Deposit RaisedTokens into the FTO<br>  - Claim the LP tokens from the FTO corresponding to their share<br>  - Claim the LaunchToken allocated to them | **Contract achieves this functionality.** |
| **YexFTOFactory.sol**<br>- Factory that generates FTOPair<br>- Allows the whitelisted addresses to:<br>  - Create Launch Token and FTOPair<br>- Allows the owner to:<br>  - Add a new token used for investment<br>  - Remove a token used for investment<br>  - Pause/resume the contract<br>  - Withdraw fees | **Contract achieves this functionality.** |
| **YexFTOLaunchToken.sol**<br>- Launch Token contract<br>- Allows the Minter role to mint tokens<br>- Allows the Burner role to burn tokens | **Contract achieves this functionality.** |
| **YexFTOPair.sol**<br>- Manages the token pair transactions<br>- Allows users to:<br>  - Deposit RaisedToken<br>  - Withdraw their deposited RaisedToken<br>  - Claim their part of the remaining LaunchToken after successful fundraising if they are a depositor | **Contract achieves this functionality.** |

| | |
|---|---|
| - Claim LP tokens after fundraising is completed<br>- Allows the factory contract to:<br>- Pause/resume the contract | |
| **BurnableHook.sol**<br>- Abstract contract<br>- Projects can use this hook to burn their own tokens when they remove liquidity from the pool | **Contract achieves this functionality.** |
| **NormalHook.sol**<br>- Abstract contract<br>- The base hook contract that all custom hooks used in the FTO Launchpad must inherit<br>- Developers should inherit from this contract when developing hooks | **Contract achieves this functionality.** |
| **VestingHook.sol**<br>- Abstract contract<br>- Projects that use this hook at launch can set up a predefined vesting schedule | **Contract achieves this functionality.** |

# Code Quality

This audit scope involves the smart contracts of the Honeypot Finance project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was required.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

# Audit Resources

We were given the Honeypot Finance projects smart contract code in the form of a Zip file.

As mentioned above, code parts are well-commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments help us understand the overall architecture of the protocol.

# Dependencies

Per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry-standard open-source projects.
Apart from libraries, its functions are used in external smart contract calls.

# Severity Definitions

| Significance | Description |
|---|---|
| High | High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community. |
| Medium | Medium-level difficulties should be solved before deployment, but won't result in loss of funds. |
| Low | Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future. |
| Gas | Gas Optimisations, issues, and inefficiencies |

# Audit Findings

## High

### [H-01] YexFTOPair - No incentives to deposit early into a pair. Users can deposit right before the endTime to earn tokens

**Description**

There are no incentives in the protocol for early depositors. Users can wait till the endTime to assess the pair and deposit. These users will earn as much rewards as early depositors. This will lead to no user depositing early which can break protocol functionality.

**Vulnerability Details**

The deposit function in the YexFTOFacede contract allows users to deposit raisedToken into the relevant pair.

```
function deposit(
    address raisedToken,
    address launchedToken,
    uint256 raisedTokenAmount
) external override {
    _deposit(
        raisedToken,
        launchedToken,
        raisedTokenAmount
    );
}
```

This allows users to earn LP tokens and launchedToken after the fundraising period ends, depending on the amount of tokens they have deposited. How much a user has earned is calculated with the _calculateLPAmount and _calculateLaunchedTokenAmount functions shown below:

```
    function _calculateLPAmount(
        address caller
    ) internal view returns (uint256 lpAmount) {
        //comments
                uint256  cumulativeLP  =  IERC20(lpToken).balanceOf(address(this))  +
totalClaimedLp;
        //comments
        lpAmount = cumulativeLP >> 1;


        //comments
        if (launchedTokenProvider != caller) {
            lpAmount =
                (raisedTokenDeposit[caller] * lpAmount) /
                depositedRaisedToken;
        }
    }
    function _calculateLaunchedTokenAmount(
        address caller
    ) internal view returns (uint256 amount) {
        //comments
        amount =
            (raisedTokenDeposit[caller] * poolLaunchedTokenAmount) /
            depositedRaisedToken;
    }
```

As observed in these functions, the amount of tokens a user can get is only dependent on their raisedTokenDeposit amount compared to the whole pool. This allows users to deposit tokens right before the endTime and be entitled to as many tokens as the early depositors.

**Proof of Concept**

Implement and run the following test in hookFTO.ts. Observe the logs to notice the vulnerability.

```
  it.only("test CustomHook launched FTO burn deposit late", async function () {
```

```
const name = "TestToken";
const symbol = "TT";

const amount = ethers.utils.parseUnits("1", 18);
const poolHandler = henloDexRouter.address;
const raisingCycle = 120; // 120 seconds
const launchedPercent = 95;

const hookPercent = 50; // lock 50% lp

let startTimestamp =
  (await ethers.provider.getBlock(ethers.provider.blockNumber)).timestamp +
  raisingCycle; // after raised
const durationSeconds = 10000; // vesting duaration

const hook_params = ethers.utils.defaultAbiCoder.encode(
  ["uint64", "uint64", "address"],
  [startTimestamp, durationSeconds, owner.address]
);

const data = ethers.utils.defaultAbiCoder.encode(
  ["uint256", "bytes"],
  [hookPercent, hook_params]
);

// 1. customHook createFTO
await customHook.createFTO(
  usdt.address,
  name,
  symbol,
  amount,
  launchedPercent,
  poolHandler,
  raisingCycle,
```

```
      data
    );

    const ftoPair_addr = await ftoFactory.allPairs(0);
    const YexFTOPair = await ethers.getContractFactory("YexFTOPair");
    const ftoPair = YexFTOPair.attach(ftoPair_addr) as YexFTOPair;
    const launchedToken = await ftoPair.launchedToken();
    const raisedTokenGetter = await customHook.raisedTokenReceiver(ftoPair_addr);
    const launchedTokenProvider = await ftoPair.launchedTokenProvider();

    expect(await ftoPair.percent4hook()).to.be.equal(hookPercent);
    expect(await customHook.beneficiary(ftoPair_addr)).to.be.equal(
      ftoPair_addr
    );
    expect(await customHook.start(ftoPair_addr)).to.be.equal(startTimestamp);
    expect(await customHook.duration(ftoPair_addr)).to.be.equal(
      durationSeconds
    );

    // 2. deposit and perform addr2
    const deposit_amount = ethers.utils.parseUnits("10", 18);
    await usdt.connect(addr2).faucet();
    await usdt.connect(addr2).approve(ftoFacade.address, deposit_amount);

    await ftoFacade
      .connect(addr2)
      .deposit(usdt.address, launchedToken, deposit_amount);

    // increase time to end of fundRaising period
    await network.provider.send("evm_increaseTime", [raisingCycle - 10]);
    await network.provider.send("evm_mine");

    // addr1 deposit
    const deposit_amount2 = ethers.utils.parseUnits("100", 18);
```

```
        await usdt.connect(addr1).faucet();
        await usdt.connect(addr1).approve(ftoFacade.address, deposit_amount2);

        await ftoFacade
          .connect(addr1)
          .deposit(usdt.address, launchedToken, deposit_amount2);

        //increase time to end the fundRaising period and perform upkeep
        await network.provider.send("evm_increaseTime", [11]);
        await network.provider.send("evm_mine");
        expect((await ftoPair.checkUpkeep("0x"))[0]).to.be.true;
        await ftoPair.performUpkeep("0x");

        // 3. Logs
        await customHook.connect(addr1).withdrawRaisedToken(ftoPair_addr);
                                    console.log("Claimable    LP    addr1:",    await
ftoFacade.connect(addr1).claimableLP(usdt.address, launchedToken));
                                    console.log("Claimable    LP    addr2:",    await
ftoFacade.connect(addr2).claimableLP(usdt.address, launchedToken));
                               console.log("Claimable   Launched   of   addr1:",   await
ftoFacade.connect(addr1).claimableLaunchedToken(usdt.address, launchedToken));
                               console.log("Claimable   Launched   of   addr2:",   await
ftoFacade.connect(addr2).claimableLaunchedToken(usdt.address, launchedToken));
    });
```

This test will print the following logs:

```
Claimable LP addr1: BigNumber { value: "2207135896050889450" }
Claimable LP addr2: BigNumber { value: "220713589605088945" }
Claimable Launched of addr1: BigNumber { value: "45454545454545454" }
Claimable Launched of addr2: BigNumber { value: "4545454545454545" }
```

**Impact**

There are no incentives to deposit early into a pair's active time. Users can deposit at the end and benefit from rewards. This will create a situation in which most users will

wait until the endTime to assess the pair. Having no users deposit early can break the protocol's functionality.

**Recommendation**

Implement a time-weighted system when calculating the amount of tokens a user has earned. Depositing early should allow users to receive more tokens than late depositors. An example solution could be to split the fundraising period into stages and implement different multipliers for these stages. For example, a 30-day fundraising period can be split into 10 stages each lasting 3 days.

Days 0-3 have a multiplier of 1
Days 3-6 have a multiplier of 0.9
...
Days 27-30 have a multiplier of 0.1

Create a new variable to keep track of users' multipliers depending on the stage they have deposited in and use this multiplier to decide on the amount of tokens they should earn.

**Note**

The Honeypot Finance team recognizes the potential imbalance that could arise if early depositors receive disproportionate benefits, putting later participants at a disadvantage. To address this, the platform is intentionally designed to ensure fairness and equity for all depositors, regardless of when they join the pool.

**Status**

Resolved

# Medium

## [M-01] YexFTOPair::refundRaisedToken - Users are not removed as an event participant after withdrawing their raised tokens

### Description

Users are added as event participants when they deposit raisedToken into a pair. However, when users withdraw their tokens, they are not removed as event participants. This will lead to a situation where users who are no longer funding the pair will benefit as an event participant.

### Vulnerability Details

The deposit function shown below, allows users to deposit raisedToken into a pair.

```
function deposit(
     address raisedToken,
     address launchedToken,
     uint256 raisedTokenAmount
 ) external override {
     _deposit(
         raisedToken,
         launchedToken,
         raisedTokenAmount
     );
 }
```

This function will call the _deposit function which will call the depositRaisedToken function in the YexFTOPair contract. Take a look at this function:

```
function depositRaisedToken(
     address depositor,
     uint256 amount
 ) external override whenNotPaused {
     //rest of the function
```

```
        /**
         * The addEvent function in the FTOFactory updates the storage variable
         *  to reflect that the depositor has participated in this FTOPair.
         */
        IYexFTOFactory(factory).addEvent(depositor, address(this));

        emit DepositRaisedToken(depositor, amount);
    }
```

However, when the pair is paused and this user withdraws their tokens with the refundRaisedToken function shown below:

```
    function refundRaisedToken() external override lock whenPaused {
        // Verify that msg.sender is a valid address that had deposited RaisedToken.
        uint256 deposit_amount = raisedTokenDeposit[msg.sender];
        require(deposit_amount > 0, "refundable amount is 0");

        raisedTokenDeposit[msg.sender] = 0;
        depositedRaisedToken -= deposit_amount;

        IERC20(raisedToken).transfer(msg.sender, deposit_amount);

        emit Refund(msg.sender, deposit_amount);
    }
```

It is observed that this user will not be removed as an event participant.

Take a look at the addEvent function in the YexFTOFactory contract below:

```
    function addEvent(address depositor, address ftoPair) external override {
        require(
            IYexFTOPair(ftoPair).raisedTokenDeposit(depositor) != 0,
            "Not participate in this rasing."
        );
        if (events_map[depositor][ftoPair] == false) {
            events_map[depositor][ftoPair] = true;
```

```
            eventParticipants[depositor].push(ftoPair);
        }
    }
```

It is observed in the highlighted part that users who are not depositors should not be event participants. This logic does not hold up when users withdraw their tokens.

**Impact**

Users that withdraw their raisedToken from the pair will stay as event participants even though they no longer have any deposited tokens in the pair.

**Recommendation**

Remove users from the mappings and arrays that are relevant to the events when they withdraw their tokens from the pair. As an example, implement a new function and an event shown below and call this function in the refundRaisedToken function.

```
// Event to be emitted when a user is removed from an event
event EventRemoved(address indexed depositor, address indexed ftoPair);

function removeEvent(address depositor, address ftoPair) external override {
    require(
        events_map[depositor][ftoPair] == true,
        "User is not a participant in this event."
    );

    // Remove the mapping
    events_map[depositor][ftoPair] = false;

    // Find and remove the ftoPair from the eventParticipants array
    uint256 length = eventParticipants[depositor].length;
    for (uint256 i = 0; i < length; i++) {
        if (eventParticipants[depositor][i] == ftoPair) {
            // Replace the found element with the last element
               eventParticipants[depositor][i] = eventParticipants[depositor][length -
1];
```

```
            // Remove the last element
            eventParticipants[depositor].pop();
            break;
        }
    }

    emit EventRemoved(depositor, ftoPair);
}
```

**Status**

Resolved

### [M-02] YexFTOFactory::addEvent - Lack of checks leads to user manipulation of the event mappings and arrays

**Description**

The addEvent function is called by a pair contract when a user deposits raisedToken into that pair. This function lacks proper checks to make sure that the pair inputted into this function is a valid pair.

**Vulnerability Details**

The addEvent function shown below, manages the events when a user deposits into a pair contract.

```
   function addEvent(address depositor, address ftoPair) external override {
        require(
            IYexFTOPair(ftoPair).raisedTokenDeposit(depositor) != 0,
            "Not participate in this rasing."
        );
        if (events_map[depositor][ftoPair] == false) {
            events_map[depositor][ftoPair] = true;
            eventParticipants[depositor].push(ftoPair);
        }
    }
```

As observed in the highlighted part, the only check done is checking the raisedTokenDeposit amount of the depositor in that pair. This means that any user can deploy a fake pair contract that will be compatible with the IYexFTOPair interface and set their raisedTokenDeposit amount in this contract to be greater than 0.
These users can now call the addEvent function with their deployed fake pair contract as the parameter and become an event participant for that fake pair.

```
function events(
    address depositor
) external view override returns (address[] memory pairs) {
    return eventParticipants[depositor];
}
```

As observed in the events function above this function will return all the fake pair contracts as if the user has participated in actual events. Depending on the off-chain use of events, users can manipulate statistics and leaderboards.

**Impact**

Users can manipulate the events mappings and arrays as they wish. This can lead to off-chain systems using events reading incorrect data.

**Recommendation**

Implement proper checks to make sure that events can only be added by legitimate pair contracts. This can be done by implementing a new modifier that checks for a mapping. Pairs launched through the YexFTOFactory contract should be added to this mapping and this modifier should act as access control for addEvent function.
An alternative change could be to change the addEvent function to receive two parameters: raisedToken and launchedToken. With these parameters, the function can use these values to get the pair corresponding to these tokens.

**Status**

Resolved

# Low

### [L-01] YexFTOPair::refundRaisedToken - Return values of transfer() not checked

**Description**

Not all ERC20 implementations revert when there's a failure in transfer(). The function signature has a boolean return value and they indicate errors that way instead. By not checking the return value, operations that should have been marked as failed may potentially go through without actually transferring anything.

**Recommendation**

To ensure the reliability and security of token transfers, it's crucial to check the return values of the transfer() function. This function often returns a boolean value indicating the success or failure of the transfer operation. By checking this return value, you can accurately determine whether the transfer was successful and handle any potential errors or failures accordingly.

**Status**

Resolved

### [L-02] YexFTOFacade, NormalHook - Missing checks for address(0) in constructors

**Description**

No address(0) checks in the constructor could lead to breaking contract functionality in case of accidental inputs.

**Recommendation**

Implement address(0) checks in relevant constructors. An example is shown below:

```
contract YexFTOFacade is IYexFTOFacade, Ownable {
    address public immutable override factory;
```

```
    constructor(address _factory) {
        require (_factory != address(0), "Address 0");
        factory = _factory;
    }
```

**Status**

Resolved

### [L-03] YexFTOPair::_perform - Unsafe ERC20 operation approve()

**Description**

Approve call does not handle non-standard ERC20 behaviour. Use safeApprove instead of approve.

Some token contracts do not return any value.

Some token contracts revert the transaction when the allowance is not zero.

**Recommendation**

Use safeApprove instead of approve

**Status**

Resolved

### [L-04] YexFTOLaunchToken - Use Ownable2step instead of Ownable

**Description**

Using Ownable2Step instead of the standard Ownable contract adds a critical layer of security to smart contracts by ensuring that ownership transfers are intentional and verified by both parties

**Recommendation**

Instead of ownable use ownable2step

**Status**

Resolved

# Gas

## [G-01] BurnableHook::_setBurnableHookParam - Prevent setting a state variable with the same value

**Description**

It is wasteful in terms of gas to set a state variable with the same value.

```
function _setBurnableHookParam(BurnableHookParam memory params) internal {
     require(params.receiver != address(0), "Receiver is invalid.");
     raisedTokenReceiver[msg.sender] = params.receiver;
 }
```

**Recommendation**

Implement checks in your function to avoid setting state variables to their existing values. Prior to updating a state variable, compare the new value with the current value and proceed with the assignment only if they differ. An example is shown below:

```
function _setBurnableHookParam(BurnableHookParam memory params) internal {
     require(params.receiver != address(0), "Receiver is invalid.");
     require(params.receiver != raisedTokenReceiver[msg.sender], "Same value");
     raisedTokenReceiver[msg.sender] = params.receiver;
 }
```

**Status**

Acknowledged

### [G-02] YexFTOPair::pause, resume - Unnecessary event field

**Description**

block.timestamp is added to the event information by default, so adding it manually will waste additional gas.

```
function pause() external override {
    //rest of the function
    emit Paused(block.timestamp);
 }
 function resume() external override {
    //rest of the function
    emit Resumed(block.timestamp);
 }
```

**Recommendation**

Remove block.timestamp from the parameters of emitted events. Instead of manually adding this field, rely on the Ethereum blockchain's inherent inclusion of this information within the block context.

**Status**

Resolved

### [G-03] YexFTOPair, YexFTOFactory, VestingHook - Revert string size optimisation

**Description**

Shortening the revert strings to fit within 32 bytes will decrease deployment time and decrease runtime Gas when the revert condition is met. Revert strings that are longer than 32 bytes require at least one additional mstore, along with additional overhead to calculate memory offset, etc.

**Recommendation**

To optimise gas usage, it is recommended to keep revert strings as short as possible and to ensure that they fit within 32 bytes. It is possible to use abbreviations or simplified error messages to keep the string length short. Doing so can reduce the amount of Gas used during deployment and runtime when the revert condition is met.

**Status**

Resolved

### [G-04] Contracts - Use custom errors in Solidity for gas efficiency

**Description**

Every unique string used as a revert reason consumes gas, making transactions more expensive. Custom errors, on the other hand, are identified by their name and the types of their parameters only, and they do not have the overhead of string storage. This means that, when using custom errors instead of require statements with string messages, the gas consumption can be significantly reduced, leading to more gas-efficient contracts.

**Recommendation**

It is recommended to use custom errors instead of reverting strings to reduce gas costs, especially during contract deployment. Custom errors can be defined using the error keyword and can include dynamic information. An example is shown below:

```
function _deposit( //YexFTOFacade.sol
     address raisedToken,
     address launchedToken,
     uint256 raisedTokenAmount
 ) internal {
     if (raisedTokenAmount <= 0){
         revert InvalidInputAmount();
     }
     //rest of the function
```

```
    }
```

**Status**

Resolved

### [G-05] VestingHook::release - Multiple accesses of a mapping should use a local variable cache

**Description**

The instances below point to the second access of a value inside a mapping within a function. Caching a mapping's value in a local storage or calldata variable when the value is accessed multiple times, saves ~42 gas per access due to not having to recalculate the key's keccak256 hash (Gkeccak256 - 30 gas), and that calculation's associated stack operations. Caching an array's struct avoids recalculating the array offsets into memory/calldata

```
function release(address ftoPair) public virtual {
     uint256 amount = releasable(ftoPair);
     _erc20Released[getPair[ftoPair].lpToken] += amount;
     emit ERC20Released(getPair[ftoPair].lpToken, amount);
     TransferHelper.safeTransfer(
         getPair[ftoPair].lpToken,
         beneficiary(ftoPair),
         amount
     );
 }
```

**Recommendation**

When a mapping value is accessed multiple times within a function, cache it in a local storage or calldata variable. This approach minimises the gas cost. An example is shown below:

```
function release(address ftoPair) public virtual {
     uint256 amount = releasable(ftoPair);
```

```
        address cachedLpToken = getPair[ftoPair].lpToken;
        _erc20Released[cachedLpToken] += amount;
        emit ERC20Released(cachedLpToken, amount);
        TransferHelper.safeTransfer(
            cachedLpToken,
            beneficiary(ftoPair),
            amount
        );
    }
```

**Status**

Resolved

### [G-06] YexFTOPair::_perform - State variables that are used multiple times in a function should be cached

**Description**

When performing multiple operations on a state variable in a function, it is recommended to cache it first. Either multiple reads or multiple writes to a state variable can save gas by caching it on the stack.

```
        function _perform()  //State variables raisedToken, launchedToken,
depositRaisedToken, depositLaunchedToken and otherPool can be cached
```

**Recommendation**

Cache state variables in stack or local memory variables within functions when they are used multiple times.

**Status**

Resolved

### [G-07] YexFTOPair::feePercent - Missing constant modifier for state variable

**Description**

State variables that are never reassigned after their initial assignment should typically be declared with the constant modifier. This ensures that these variables remain unmodified and communicate their unchanging nature. Using the constant modifier also offers potential gas savings, as accessing these variables does not require any storage operations.

**Recommendation**

Declare state variables that are not reassigned after initialization with the 'constant' modifier in Solidity.

**Status**

Resolved

### [G-08] Contracts - Functions guaranteed to revert when called by normal users can be marked payable

**Description**

If a function modifier such as onlyOwner is used, the function will revert if a normal user tries to pay the function. Marking the function as payable will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided.

```
function addRaisedToken(address _raisedToken) external onlyOwner
function removeRaisedToken(address _raisedToken) external onlyOwner {
function mint(address to, uint256 amount) public onlyRole(MINTER_ROLE)
function burn(uint256 amount) public onlyRole(BURNER_ROLE) {
```

**Recommendation**

It is recommended to make these functions payable

**Status**

Resolved

## [G-09] YexFTOFactory::_createPair - No check on return address of create2

**Description**

The createFTO function calls out to _createPair function which uses openzeppelin create2 opcode but there isn't any kind of check on the return price of create2.

**Vulnerability Details**

The create2 opcode when deploying the contract always returns the address of the contract but the code isn't checking the return value and if for any reason the deployment fails it will return address 0.

```
    assembly {
            pair := create2(0, add(bytecode, 32), mload(bytecode), salt)

        YexFTOLaunchToken(launchedToken).mint(pair, launchedTokenSupply);
```

**Impact**

The function might not work as intended and will revert

**Recommendation**

It is recommended to check for the return address of create2 opcode as OpenZeppelin is doing in this official code.

**Status**

Resolved

# Centralisation

The Honeypot Finance project values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.

# Conclusion

After Hashlocks analysis, the Honeypot Finance project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

# Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

**Manual Code Review:**

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

**Vulnerability Analysis:**

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

**Documenting Results:**

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we still need to verify the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown not to represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

**Suggested Solutions:**

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

# Disclaimers

## Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

## Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

# About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

**Website:** hashlock.com.au
**Contact:** info@hashlock.com.au


#Hashlock.