Kayen Security Review

Version 2.0

27.07.2024

Conducted by:

MaslarovK, Security Researcher

radev-eth, Security Researcher

Table of Contents

1 About MaslarovK

MaslarovK is a security researcher from Bulgaria. Co-Founder of Rezolv Solutions.

2 About radev.eth

radev_eth is a security researcher from Bulgaria. Co-Founder of Rezolv Solutions.

3 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

4 Risk classification

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

4.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

4.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

4.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

5 Executive summary

Overview

Project Name	Spectra
Repository	https://github.com/Kayen-Protocol/KayenswapV2.5
Commit hash	87f3e1519cd503318af3821862272ef87896157b
Resolution	f7e1634958060f5e362eecd47b8c612e099e9eb8
Documentation	N/A
Methods	Manual review & testing

Scope

src/KayenMasterRouterV2.sol
src/KayenPair.sol

Issues Found

Critical risk	0
High risk	0
Medium risk	6
Low risk	8
Informational	0

6 Findings

6.1 Medium risk

7 [M-01] `KayenPair.sol#_update()` - `timeElapsed` is designed to overflow, but it cannot because of used solidity version

7.1 Summary

In the `KyberPair` contract, the `_update` function relies on arithmetic overflow for `timeElapsed` and `priceCumulative` calculations. This design, inherited from UniswapV2, intentionally allows overflow. However, `KyberPair` contract uses `Solidity >=0.8.0`, where such overflows cause automatic reverts, potentially leading to a permanent denial of service (DoS) in the pool. This could disable core functions (mint, burn, swap), locking all funds in the contract.

- `KyberPair.sol#_update()` function:

```
// update reserves and, on the first call per block, price accumulators
function _update(uint256 balance0, uint256 balance1, uint112 _reserve0, uint112
    _reserve1) private {
    if (balance0 > type(uint112).max || balance1 > type(uint112).max) revert
        Overflow();
    uint32 blockTimestamp = uint32(block.timestamp % 2 ** 32);
    uint32 timeElapsed = blockTimestamp - blockTimestampLast; // overflow is
        desired
    if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
        // * never overflows, and + overflow is desired
        price0CumulativeLast += uint256(UQ112x112.encode(_reserve1).uqdiv(
            _reserve0)) * timeElapsed;
        price1CumulativeLast += uint256(UQ112x112.encode(_reserve0).uqdiv(
            _reserve1)) * timeElapsed;
    }
    reserve0 = uint112(balance0);
    reserve1 = uint112(balance1);
    blockTimestampLast = blockTimestamp;
    emit Sync(reserve0, reserve1);
}
```

As we can see that there is even a comment on this line that specifically states that overflow is desired. The code in UniswapV2 looks exactly the same, with one crucial exception. UniswapV2 uses `Solidity 0.5.16`, where overflows could occur and would not trigger a panic revert.

To see how `timeElapsed` can revert, we have to take a look at how `blockTimestamp` is calculated.

`blockTimestamp = uint32(block.timestamp % 2 ** 32);`

To understand why we get the remainder from `2 ** 32` you can read this short explanation.

Knowing this, when `block.timestamp > 2 ** 32`, the remainder is calculated with `2 ** 32`, the idea is to basically wrap back around to 0 when `block.timestamp > 2 ** 32`

This means that when `block.timestamp > 2 ** 32`, the remainder is set for `blockTimestamp` which is then used in the calculation for `timeElapsed`

```
timeElapsed = blockTimestamp - blockTimestampLast; // overflow is desired
```

Know that we know that `blockTimestamp` can wrap back around to 0, we will have a situation where `blockTimestampLast` is bigger than `blockTimestamp` and thus, an underflow will occur. When this happens any call to `KyberPair.sol#_update()` function will revert, basically bricking the entire protocol.

7.2 Impact

The bug can lead to a permanent DoS for the pool, rendering all core functions inoperable and locking user funds indefinitely. Consequently, all funds would be locked within the contract. Despite its low probability due to the extended timeframe for the event's occurrence, it has a high impact once it occurs.

7.3 Tools Used

- Manual Code Review

7.4 Recommended Mitigation Steps

Wrap the calculation for `timeElapsed` in an unchecked block to prevent automatic reverts:

```
    // update reserves and, on the first call per block, price accumulators
    function _update(uint256 balance0, uint256 balance1, uint112 _reserve0, uint112
        _reserve1) private {
        if (balance0 > type(uint112).max || balance1 > type(uint112).max) revert
            Overflow();
        uint32 blockTimestamp = uint32(block.timestamp % 2 ** 32);
-       uint32 timeElapsed = blockTimestamp - blockTimestampLast; // overflow is
    desired
+       unchecked {
+           uint32 timeElapsed = blockTimestamp - blockTimestampLast; // overflow is
     desired
+       }
        if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
            // * never overflows, and + overflow is desired
            price0CumulativeLast += uint256(UQ112x112.encode(_reserve1).uqdiv(
                _reserve0)) * timeElapsed;
            price1CumulativeLast += uint256(UQ112x112.encode(_reserve0).uqdiv(
                _reserve1)) * timeElapsed;
        }
        reserve0 = uint112(balance0);
        reserve1 = uint112(balance1);
        blockTimestampLast = blockTimestamp;
        emit Sync(reserve0, reserve1);
    }
```

Resolution: Fixed

8 [M-02] `KayenMasterRouterV2.sol#_addLiquidity()` - Users are Unduly Penalized with High Gas Costs for Transactions that Fail due to variable conditions

8.1 Summary

In the `KayenMasterRouterV2.sol` contract, the `_addLiquidity()` function uses `assert` to check that the optimal amount of `tokenA` to be added (`amountAOptimal`) does not exceed the desired amount (`amountADesired`). This assertion depends on user inputs and current state variables which can fluctuate. The use of `assert` here is inappropriate because if the condition fails, it consumes all remaining gas, penalizing users with high gas costs.

- `KayenMasterRouterV2.sol#_addLiquidity()` function:

```
function _addLiquidity(
    address tokenA,
    address tokenB,
    uint256 amountADesired,
    uint256 amountBDesired,
    uint256 amountAMin,
    uint256 amountBMin
) internal virtual returns (uint256 amountA, uint256 amountB) {
    // create the pair if it doesn't exist yet
    if (IKayenFactory(factory).getPair(tokenA, tokenB) == address(0)) {
        IKayenFactory(factory).createPair(tokenA, tokenB);
    }
    (uint256 reserveA, uint256 reserveB) = KayenLibrary.getReserves(factory,
        tokenA, tokenB);
    if (reserveA == 0 && reserveB == 0) {
        (amountA, amountB) = (amountADesired, amountBDesired);
    } else {
        uint256 amountBOptimal = KayenLibrary.quote(amountADesired, reserveA,
            reserveB);
        if (amountBOptimal <= amountBDesired) {
            if (amountBOptimal < amountBMin) revert InsufficientBAmount();
            (amountA, amountB) = (amountADesired, amountBOptimal);
        } else {
            uint256 amountAOptimal = KayenLibrary.quote(amountBDesired, reserveB
                , reserveA);
            assert(amountAOptimal <= amountADesired);
            if (amountAOptimal < amountAMin) revert InsufficientAAmount();
            (amountA, amountB) = (amountAOptimal, amountBDesired);
        }
    }
}
```

8.2 Impact

The use of `assert` in this context leads to transactions consuming all remaining gas when the condition fails. This is especially punitive in high-gas-price environments, causing users to pay high fees for failed transactions.

8.3 Tools Used

Manual Code Review

8.4 Recommended Mitigation Steps

Replace `assert` with `require` to prevent excessive gas consumption and provide a meaningful error message:

```
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint256 amountADesired,
        uint256 amountBDesired,
        uint256 amountAMin,
        uint256 amountBMin
    ) internal virtual returns (uint256 amountA, uint256 amountB) {
        // create the pair if it doesn't exist yet
        if (IKayenFactory(factory).getPair(tokenA, tokenB) == address(0)) {
            IKayenFactory(factory).createPair(tokenA, tokenB);
        }
        (uint256 reserveA, uint256 reserveB) = KayenLibrary.getReserves(factory,
            tokenA, tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint256 amountBOptimal = KayenLibrary.quote(amountADesired, reserveA,
                reserveB);
            if (amountBOptimal <= amountBDesired) {
                if (amountBOptimal < amountBMin) revert InsufficientBAmount();
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint256 amountAOptimal = KayenLibrary.quote(amountBDesired, reserveB
                    , reserveA);
-               assert(amountAOptimal <= amountADesired);
+               require(amountAOptimal <= amountADesired, "Insufficient amountA
    desired");
                if (amountAOptimal < amountAMin) revert InsufficientAAmount();
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
```

Resolution: Fixed

9 [M-03] Incorrect Swap Executions Due to Misalignment between amounts and path

9.1 Summary

In the `KayenMasterRouterV2.sol` contract, the `_swap` function facilitates token swaps through liquidity pools represented by token pairs along a specified path. The function calculates the output amount for each swap in the path and executes these swaps sequentially. A critical assumption for this function's correct operation is the alignment between the `amounts` array and the `path` array. Specifically, for a path of N tokens, there should be N-1 swaps, and thus, the `amounts` array should contain N elements.

- `KayenMasterRouterV2.sol#_swap()` function:

```
// requires the initial amount to have already been sent to the first pair
function _swap(uint256[] memory amounts, address[] memory path, address _to)
    internal virtual {
    for (uint256 i; i < path.length - 1; i++) {
        (address input, address output) = (path[i], path[i + 1]);
        (address token0, ) = KayenLibrary.sortTokens(input, output);
        uint256 amountOut = amounts[i + 1];
        (uint256 amount0Out, uint256 amount1Out) = input == token0
            ? (uint256(0), amountOut)
            : (amountOut, uint256(0));
        address to = i < path.length - 2 ? KayenLibrary.pairFor(factory, output,
             path[i + 2]) : _to;
        IKayenPair(KayenLibrary.pairFor(factory, input, output)).swap(amount0Out
            , amount1Out, to, new bytes(0));
    }
}
```

9.2 Impact

1. If `amounts.length` is less than `path.length - 1`, the function will attempt to access an index of `amounts` that does not exist, leading to a runtime error and reverting the transaction.
2. If `amounts.length` is greater than `path.length`, some amounts will not be used, indicating a logic error or a misinterpretation of how the function should be called. Conversely, if `amounts.length` is less than `path.length`, it could result in attempting swaps without valid output amounts, leading to incorrect behavior or transaction failure.

9.3 Tools Used

Manual Code Review

9.4 Recommended Mitigation Steps

Add a validation step at the beginning of the function to ensure that `amounts.length == path.length`:

```
require(amounts.length == path.length, "Amounts and path length mismatch");
```

Resolution: Fixed

10 [M-04] Big flash loans will fail due to fee

10.1 Summary

In the `KayenPair.sol` contract, there is a potential issue with flash loans due to the implementation of flash loan fees. The function transfers the requested loan amount to the recipient and immediately transfers the associated flash loan fees to the `feeTo` address before executing the callback. This sequence can cause the transaction to fail for large flash loans because the pool must have additional funds to cover the fees before the callback is executed.

```
// this low-level function should be called from a contract which performs
    important safety checks
function swap(uint256 amount0Out, uint256 amount1Out, address to, bytes calldata
     data) external lock {

        // ... other code ...

        if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); //
            optimistically transfer tokens
        if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); //
            optimistically transfer tokens
        if (IKayenFactory(factory).flashOn() && data.length > 0) {
            if (amount0Out > 0) {
                _safeTransfer(
                    _token0,
                    IKayenFactory(factory).feeTo(),
                    (amount0Out * IKayenFactory(factory).flashFee()) / 10000
                );
                amount0Out = (amount0Out * (10000 + IKayenFactory(factory).
                    flashFee())) / 10000;
            }
            if (amount1Out > 0) {
                _safeTransfer(
                    _token1,
                    IKayenFactory(factory).feeTo(),
                    (amount1Out * IKayenFactory(factory).flashFee()) / 10000
                );
                amount1Out = (amount1Out * (10000 + IKayenFactory(factory).
                    flashFee())) / 10000;
            }
            IKayenCallee(to).KayenCall(msg.sender, amount0Out, amount1Out, data)
                ;
        }
```

```
        // ... other code ...

    }
```

10.2 Impact

Flash loans of large amounts may fail due to insufficient funds to cover both the loan and the immediate transfer of the flash loan fees. This can prevent users from utilizing the flash loan functionality for large amounts, limiting the utility of the contract.

10.3 Tools Used

Manual Code Review

10.4 Recommended Mitigation Steps

To ensure large flash loans can be processed successfully, transfer the requested amount to the recipient, execute the callback, and then transfer the associated fees. This sequence allows the recipient to return the loan amount plus fees within the callback, ensuring the pool has sufficient funds to cover the fees.

1. Transfer the requested amount to the recipient.
2. Execute the callback.
3. The recipient returns the requested amount and the flash loan fees.
4. Transfer the fees to the `feeTo` address.

Resolution: Fixed

11 [M-05] The burning of small liquidity will be revert

11.1 Summary

In the `KayenPair.sol` contract, there is an issue related to burning small amounts of liquidity. In the `burn` function, users burn liquidity to receive corresponding amounts of `token0` and `token1`. However, the check for zero amounts of `amount0` or `amount1` prevents the burning of small liquidity amounts.

- “ function:

```
    // this low-level function should be called from a contract which performs
        important safety checks
    function burn(address to) external lock returns (uint256 amount0, uint256
        amount1) {
```

```
        // ... other code ...

        uint256 _totalSupply = totalSupply; // gas savings, must be defined here
            since totalSupply can update in _mintFee
        amount0 = (liquidity * balance0) / _totalSupply; // using balances ensures
            pro-rata distribution
        amount1 = (liquidity * balance1) / _totalSupply; // using balances ensures
            pro-rata distribution
        if (amount0 == 0 || amount1 == 0) revert InsufficientLiquidityBurned();

        // ... other code ...

    }
```

For example:

Alice uses `1e7*1e18 token0 and 1e1*1e18` token1 to create pool. So the totalSupply of pool will be `1e4*1e18`. `(reserve0 = 1e7*1e18, reserve1 = 1e1*1e18)`

Then Bob mints liquidity using `1e7 token0` and `1e1 token1`. So the Bob's liquidity will be 1e4.

Finally, if Bob wants to burn liquidity which is smaller than `1000`, his request will be reverted as the `amount1 is 0`. However, the `amount0 is 999000`.

11.2 Impact

Users cannot burn small amounts of liquidity, which may lead to fund locks. After multiple burns, users may have small amounts of liquidity remaining that they cannot burn, resulting in inaccessible funds.

11.3 Tools Used

Manual Code Review

11.4 Recommended Mitigation Steps

Change the logic from `OR` to `AND` to allow burning of small liquidity amounts unless both `amount0` and `amount1` are zero:

```
    // this low-level function should be called from a contract which performs
        important safety checks
    function burn(address to) external lock returns (uint256 amount0, uint256
        amount1) {

        // ... other code ...

        uint256 _totalSupply = totalSupply; // gas savings, must be defined here
            since totalSupply can update in _mintFee
        amount0 = (liquidity * balance0) / _totalSupply; // using balances ensures
            pro-rata distribution
```

```
        amount1 = (liquidity * balance1) / _totalSupply; // using balances ensures
            pro-rata distribution
-       if (amount0 == 0 || amount1 == 0) revert InsufficientLiquidityBurned();
+       if (amount0 == 0 && amount1 == 0) revert InsufficientLiquidityBurned();

        // ... other code ...

    }
```

Resolution: Fixed

12 [M-06] `KayenMasterRouterV2` will not be compatible with `ERC20s` which do not return bool on their `approve`function

12.1 Summary

`KayenMasterRouterV2` will not be compatible with ERC20s which do not return bool on their `approve` function.

Some ERC20s do not return a `bool` value on their `approve` function (e.g. USDT). Since the used interface expects a bool method to be returned, it will be impossible for the router to work with tokens that do not return any value.

```
    function _approveAndWrap(address token, uint256 amount) private returns (address
         wrappedToken) {
        uint256 allowance = IERC20(token).allowance(address(this), wrapperFactory);
        if (allowance < amount) {
            // Approve the maximum possible amount
            IERC20(token).approve(wrapperFactory, type(uint256).max);
        }
        wrappedToken = IChilizWrapperFactory(wrapperFactory).wrap(address(this),
            token, amount);
    }
```

12.2 Impact

Contract will not be able to work with tokens which do not return a `bool` on their `approve` function

12.3 Tool used

Manual Code Review

12.4 Recommendation

Use `safeApprove()` function instead of `approve()` function.

Resolution: Fixed

12.5 Low risk

13 [L-01] Unnecessary call to `_transfer` function inside `ChilizWrappedERC20#withdrawTo()` function will leads to gas Griefing

13.1 Summary

The `withdrawTo` function checks if `msg.sender != account`, then it will transfer wrapped tokens to the `account`. However, it does not check if `amount - burntAmount` is greater than zero. In this case, all calls to `_transfer` are a waste of gas.

- `ChilizWrappedERC20.sol#withdrawTo()` function:

```
function withdrawTo(address account, uint256 amount)
    public
    virtual
    returns (bool success, uint256 unwrappedAmount)
{
    if (address(underlyingToken) == address(0)) revert NotInitialized();
    uint256 unwrapAmount = amount / decimalsOffset;
    if (unwrapAmount == 0) revert CannotWithdraw();
    address msgSender = _msgSender();
    uint256 burntAmount = unwrapAmount * decimalsOffset;
    _burn(msgSender, burntAmount);
    SafeERC20.safeTransfer(underlyingToken, account, unwrapAmount);
    if (msgSender != account) {
        _transfer(msgSender, account, amount - burntAmount);
    }

    emit Withdraw(account, amount);

    return (true, unwrapAmount);
}
```

The protocol facilitates the conversion of wrapped tokens back to underlying tokens via the `ChilizWrappedERC20#withdrawTo()` function. This function can be called by `ChilizWrapperFactory:unwrap` or directly by the owner of the wrapped token.

The `withdrawTo` function converts the specified amount to underlying tokens and transfers them to the `account` address. It also checks if the `msg.sender` is not the owner account, in this case it sends the wrapped tokens to the `account` address. However, it also calls the `_transfer` function in the case where `amount - burnAmount == 0`, which does not change any balance but consumes additional gas. This leads to Griefing attack or wastes gas for end users.

13.2 Impact

Users will be paying extra gas due to missing checks.

13.3 Tool used

Manual Code Review

13.4 Recommendation

Modify `if statement` to also check for `amount-burnAmount > 0` as follows:

```
    function withdrawTo(address account, uint256 amount)
        public
        virtual
        returns (bool success, uint256 unwrappedAmount)
    {
        if (address(underlyingToken) == address(0)) revert NotInitialized();
        uint256 unwrapAmount = amount / decimalsOffset;
        if (unwrapAmount == 0) revert CannotWithdraw();
        address msgSender = _msgSender();
        uint256 burntAmount = unwrapAmount * decimalsOffset;
        _burn(msgSender, burntAmount);
        SafeERC20.safeTransfer(underlyingToken, account, unwrapAmount);
-       if (msgSender != account) {
-           _transfer(msgSender, account, amount - burntAmount);
-       }
+       if (msgSender != account && amount - burntAmount > 0 )  {
+           _transfer(msgSender, account, amount - burntAmount);
+       }

        emit Withdraw(account, amount);

        return (true, unwrapAmount);
    }
```

Resolution: Aknowledged

14 [L-02] `feeTo` address used to determine if `feeOn` should be true isn't consistent with `UniswapV2`

14.1 Summary

In the `KayenFactory.sol` contract, the `feeTo` address is used to determine if fees should be enabled (`feeOn`). The issue is that the `feeTo` address is initialized to `0x000000000000000000000000000000000000dEaD` in the constructor. This means the code mistakenly assumes the fee is enabled when `feeTo` is set to this address, which results in sending funds to an address where they are lost forever.

- **Initialization in Constructor:**

```
    address public constant DEAD = 0x000000000000000000000000000000000000dEaD;

    // ... other code ...
```

```
constructor(address _feeToSetter) {
    require(_feeToSetter != address(0), "KF: ZERO_ADDRESS");
    feeToSetter = _feeToSetter;
    flashOn = false;
    feeTo = DEAD;
    fee = 30; // 30/10000 = 0.3%
    denominatorFactor = 1;
}
```

- `KayenPair.sol#_mintFee()` function:

```
// if fee is on, mint liquidity equivalent to maximum 1/2th of the growth in
    sqrt(k)
function _mintFee(uint112 _reserve0, uint112 _reserve1) private returns (bool
    feeOn) {
    address feeTo = IKayenFactory(factory).feeTo(); // get feeTo address
    feeOn = feeTo != address(0);
    uint256 _kLast = kLast; // gas savings
    if (feeOn) {
        if (_kLast != 0) {
            uint256 rootK = Math.sqrt(uint256(_reserve0) * _reserve1);
            uint256 rootKLast = Math.sqrt(_kLast);
            if (rootK > rootKLast) {
                uint256 numerator = totalSupply * (rootK - rootKLast);
                uint256 denominator = rootK * IKayenFactory(factory).
                    denominatorFactor() + rootKLast;
                uint256 liquidity = numerator / denominator;
                // distribute LP fee
                if (liquidity > 0) _mint(feeTo, liquidity);
            }
        }
    } else if (_kLast != 0) {
        kLast = 0;
    }
}
```

14.2 Impact

The contract incorrectly assumes fees are enabled due to the `feeTo` address being set to `0x000000000000000000000000000000000000dEaD`. This results in funds being sent to an unusable address, effectively losing them forever and potentially disrupting the expected fee logic in the protocol.

14.3 Tools Used

Manual Code Review

14.4 Recommended Mitigation Steps

1. Do not initialize `feeTo` with any value (default to `address(0)`):

```
    constructor(address _feeToSetter) {
        require(_feeToSetter != address(0), "KF: ZERO_ADDRESS");
        feeToSetter = _feeToSetter;
        flashOn = false;
-       feeTo = DEAD;
        fee = 30; // 30/10000 = 0.3%
        denominatorFactor = 1;
    }
```

2. Alternatively, update the fee check to account for the `DEAD` address:

```
    feeOn = (feeTo != address(0) && feeTo != DEAD);
```

Resolution: Fixed

15 [L-03] Missing Initializer Modifier for `initialize()` Function in KayenPair and ChilizWrappedERC20 Contracts

15.1 Summary

In the `KayenPair.sol` and `ChilizWrappedERC20.sol` contracts, the `initialize` functions lack the `initializer` modifier, which is essential for proxy-based upgradeable contracts. This modifier ensures that the initializer function cannot be invoked multiple times, preventing potential reinitialization attacks.

- `KayenPair.sol#initialize()` function:

```
    // called once by the factory at time of deployment
    function initialize(address _token0, address _token1) external {
        if (msg.sender != factory) revert Forbidden();
        token0 = _token0;
        token1 = _token1;
    }
```

- `ChilizWrappedERC20.sol#initialize()` function:

```
    function initialize(IERC20 _underlyingToken) external {
        if (msg.sender != factory) revert Forbidden();
        if (_underlyingToken.decimals() >= 18) revert InvalidDecimals();
        if (address(underlyingToken) != address(0)) revert AlreadyExists();

        underlyingToken = _underlyingToken;
        decimalsOffset = 10 ** (18 - _underlyingToken.decimals());
        name = string(abi.encodePacked("Wrapped ", _underlyingToken.name()));
        symbol = string(abi.encodePacked("W", _underlyingToken.symbol()));
    }
```

15.2 Impact

Without the `initializer` modifier, these functions can be called multiple times, leading to potential reinitialization attacks and state corruption in the contract.

15.3 Tools Used

Manual Code Review

15.4 Recommended Mitigation Steps

Add the `initializer` modifier from OpenZeppelin's `Initializable` library to the `initialize` functions in both contracts to ensure they can only be called once.

```
function initialize(address _token0, address _token1) external initializer {
    if (msg.sender != factory) revert Forbidden();
    token0 = _token0;
    token1 = _token1;
}
```

```
function initialize(IERC20 _underlyingToken) external initializer {
    if (msg.sender != factory) revert Forbidden();
    if (_underlyingToken.decimals() >= 18) revert InvalidDecimals();
    if (address(underlyingToken) != address(0)) revert AlreadyExists();

    underlyingToken = _underlyingToken;
    decimalsOffset = 10 ** (18 - _underlyingToken.decimals());
    name = string(abi.encodePacked("Wrapped ", _underlyingToken.name()));
    symbol = string(abi.encodePacked("W", _underlyingToken.symbol()));
}
```

Resolution: Aknowledged

16 [L-04] Missing Contract-Existence Checks Before Low-Level Calls

16.1 Summary

In `KayenPair.sol`, the `_safeTransfer` function performs a low-level call without checking if the `token` address contains a valid contract. Low-level calls to non-contract addresses will return success, posing a security risk.

```
function _safeTransfer(address token, address to, uint256 value) private {
    (bool success, bytes memory data) = token.call(abi.encodeWithSelector(SELECTOR,
        to, value));
    if (!success || (data.length != 0 && !abi.decode(data, (bool)))) revert
        TransferFailed();
}
```

16.2 Impact

Low-level calls to non-contract addresses will return success, causing potential false positives and security issues. Funds might be transferred to incorrect addresses without proper validation.

16.3 Tools Used

Manual Code Review

16.4 Recommended Mitigation Steps

Add a check to ensure that the `token` address is a contract before performing the low-level call. For example, use the `Address.isContract` method from OpenZeppelin's library.

Resolution: Aknowledged

17 [L-05] Setters Should Prevent Re-Setting of the Same Value

17.1 Summary

In `KayenFactory.sol`, the `setFlashOn` function allows re-setting the `flashOn` state to the same value, which is unnecessary and may confuse offline parsers, especially since the function emits an event each time it is called.

```
function setFlashOn(bool _flashOn) external onlyFeeToSetter {
    flashOn = _flashOn;
    emit SetFlashOn(_flashOn);
}
```

17.2 Impact

Re-setting the same value can lead to redundant state changes and unnecessary event emissions, which may confuse offline parsers and lead to inefficient use of resources.

17.3 Tools Used

Manual Code Review

17.4 Recommended Mitigation Steps

Add a condition to check if the new value is different from the current value before performing the state change and emitting the event.

```
function setFlashOn(bool _flashOn) external onlyFeeToSetter {
    if (flashOn != _flashOn) {
        flashOn = _flashOn;
        emit SetFlashOn(_flashOn);
    }
}
```

Resolution: Aknowledged

18 [L-06] Use of `abi.encodeWithSignature/abi.encodeWithSelector` Instead of `abi.encodeCall`

18.1 Summary

In `KayenPair.sol`, the `_safeTransfer` function uses `abi.encodeWithSelector`, which is prone to typos and lacks type safety. Using `abi.encodeCall` improves code safety and readability.

```
(bool success, bytes memory data) = token.call(abi.encodeWithSelector(SELECTOR, to,
    value));
```

18.2 Impact

Using `abi.encodeWithSelector` increases the risk of typos and reduces type safety, potentially leading to runtime errors and bugs.

18.3 Tools Used

Manual Code Review

18.4 Recommended Mitigation Steps

Refactor the code to use `abi.encodeCall` for better safety and readability.

```
(bool success, bytes memory data) = token.call(abi.encodeCall(IERC20.transfer, (to,
    value)));
```

Resolution: Fixed

19 [L-07] Some Popular ERC20 Tokens Revert on Approve Larger than uint96

19.1 Summary

Certain ERC20 tokens, such as `UNI` and `COMP`, revert if the approval amount is larger than `uint96`. This issue affects the `approve` calls in `KayenMasterRouterV2.sol` and `ChilizWrapperFactory.sol`.

1. **KayenMasterRouterV2.sol**

```
IERC20(token).approve(wrapperFactory, type(uint256).max);
```

2. **ChilizWrapperFactory.sol**

```
token.approve(approveToken, amount);
```

19.2 Impact

Approving amounts larger than `uint96` for tokens like `UNI` and `COMP` will revert, leading to failed transactions and user inconvenience.

19.3 Tools Used

Manual Code Review

19.4 Recommended Mitigation Steps

Use `SafeERC20`'s `safeApprove` method from OpenZeppelin, which handles the approval logic more safely, or explicitly handle the approval amount to ensure it does not exceed `uint96`.

Resolution: Aknowledged

20 [L-08] Off-by-one issue in ensure modifier

20.1 Summary

Off-by-one issue in `KayenMasterRouterV2::ensure`.

```
    /// @notice Ensures that the transaction is executed before the deadline
    /// @dev This modifier is used to prevent pending transactions from being
       executed after a certain time
    /// @param deadline The Unix timestamp after which the transaction will revert
    /// @custom:error Expired If the current block timestamp is greater than or
       equal to the deadline
     modifier ensure(uint256 deadline) {
```

```
            if (deadline < block.timestamp) revert Expired();
        _;
 }
```

`Deadline` should be expired if it is equal to the `block.timestamp` as per the comment above.

20.2 Impact

Will allow one more block.

20.3 Tools Used

Manual Code Review

20.4 Recommended Mitigation Steps

Change the modifier as follows:

```
          modifier ensure(uint256 deadline) {
                if (deadline < block.timestamp) revert Expired();
          _;
```

Resolution: Aknowledged