[M-01] Missing Check for Existing Pair in `FFactory::createPair`

Description:
The `createPair` function does not check if a trading pair already exists before creating a new one. This could lead to conflicts when multiple pairs of the same tokens are created.

Impact:

- Potential conflicts in identifying which pair address to use in further operations.

Recommendation:
Implement a check to ensure that a trading pair does not already exist before creating a new pair.

```
function createPair(address tokenA, address tokenB) external
onlyRole(CREATOR_ROLE) nonReentrant returns (address) {
    // Check if pair already exists
    require(_pair[tokenA][tokenB] == address(0), "Pair already exists");

    // Proceed with creating pair
    address pair = _createPair(tokenA, tokenB);
    return pair;
}
```

[M-02] Lack of Slippage Protection in `Bonding::sell` and `Bonding::buy` Functions

Description:
The `sell` and `buy` functions in the `Bonding` contract enable users to trade base tokens for asset tokens. However, these functions lack slippage protection mechanisms, which can result in users paying excessive prices for assets due to market fluctuations. This exposes the contract to potential "sandwich attacks."

Scenario Example:
Assume a pool where the ratio of `memeToken` to `assetToken` is 10:1, meaning the asset token is 10 times more valuable. If a user (Alice) attempts to buy memeToken, an attacker (Bob) can exploit the lack of slippage protection by executing a sandwich attack. Bob front-runs Alice’s transaction, inflating the price of memeToken, and then back-runs Alice’s transaction, selling at a higher price, causing Alice to pay significantly more than expected.

Impact:

- Users can suffer significant financial losses due to slippage.
- Attackers can exploit the situation to profit at the user's expense.

Proof of Concept:

- The absence of a slippage control mechanism allows attackers to manipulate prices within the trading pool.

Recommendation:
Implement the `minAmountOut` parameter in the `buy` and `sell` functions to provide slippage protection, ensuring that users are not subjected to unexpected price changes.

```
function buy(uint256 amountIn, address tokenAddress) public payable returns (bool)
{
    // Existing logic here...

    // Slippage protection - use minAmountOut to prevent excessive slippage
    uint256 minAmountOut = calculateMinAmountOut(amountIn, tokenAddress);
    require(amount0Out >= minAmountOut, "Slippage too high");

    // Continue function logic...
}
```

[M-03] Missing Maximum Value Restrictions

Description:
Functions like `FERC20::updateMaxTx` and `Bonding::setFee` lack restrictions on maximum values for critical parameters, which could result in the application of unreasonable values.

Impact:

- Lack of safeguards could lead to the setting of excessive transaction limits or fees, potentially destabilizing the token economy.

Recommendation:

- Implement checks to restrict the maximum allowable values for sensitive parameters like `maxTx` and `fee`.

```
function updateMaxTx(uint256 newMaxTx) external onlyRole(ADMIN_ROLE) {
    require(newMaxTx <= MAX_TX_LIMIT, "Exceeds maximum transaction limit");
    maxTx = newMaxTx;
}
```

[M-04] Race Condition Vulnerability in `FERC20::approve` Function

Description:
The `approve` function in the `FERC20` contract is susceptible to a well-known race condition. If a user (Alice) approves a certain allowance for another user (Bob) and then attempts to reduce or modify this allowance,Bob could front-run the transaction, using the previously approved allowance before Alice's update is applied and after that can also redeem the newly approved allowance.

Impact:

- Malicious users could exploit this vulnerability to spend more tokens than the user intended.

Recommendation:

- Use the `increaseAllowance` or `decreaseAllowance` functions, which modify allowances atomically to prevent front-running.
- Alternatively, reset the allowance to 0 before updating to a new value.

```
function approve(address spender, uint256 amount) public returns (bool) {
    uint256 currentAllowance = allowance[msg.sender][spender];

    // Mitigate race condition: reset allowance to 0 first before changing to new
amount
    if (currentAllowance != 0) {
        _approve(msg.sender, spender, 0);
    }

    _approve(msg.sender, spender, amount);
    return true;
}
```

[L-01] Missing Token Address Validation in `FFactory::createPair`

Description:
The `createPair` function does not check whether the two provided token addresses are different. This could allow the creation of a pool where both tokens are the same address, making the pool ineffective.

Impact:

- The creation of a token pair where both tokens are identical undermines the intended purpose of an Automated Market Maker (AMM).

Recommendation:

- Implement a validation check to ensure that tokenA and tokenB are not the same address.

```
function createPair(address tokenA, address tokenB) external
onlyRole(CREATOR_ROLE) nonReentrant returns (address) {
    // Check if tokens are the same
    require(tokenA != tokenB, "Tokens should be different");

    // Proceed with creating the pair
    address pair = _createPair(tokenA, tokenB);
    return pair;
}
```

[L-02] Missing Input Validation in Multiple Functions

Description:
Several functions in the protocol lack input validation, especially checks for zero values. This could lead to unintended behaviors and vulnerabilities.

Impact:

- Functions could execute with invalid or unintentional inputs, potentially causing issues or unexpected behavior.

Recommendation:
Add necessary checks for zero inputs across multiple functions, such as `FFactory::initialize` and `FRouter::sell`.

```
function initialize(address taxVault_, uint256 buyTax_, uint256 sellTax_) external
initializer {
    // Ensure non-zero taxVault address
    require(taxVault_ != address(0), "Zero taxVault address");

    taxVault = taxVault_;
    buyTax = buyTax_;
    sellTax = sellTax_;
}
```

```
function sell(uint256 amountIn, address tokenAddress, address to) public
nonReentrant onlyRole(EXECUTOR_ROLE) returns (uint256, uint256) {
    // Ensure no zero addresses or amounts
    require(tokenAddress != address(0), "Zero addresses are not allowed");
    require(to != address(0), "Zero addresses are not allowed");
    require(amountIn != 0, "Zero Amount In");

    // Logic for sell
}
```

[L-03] Stuck ETH in `Bonding::buy` Function

Description:
The `Bonding::buy` function is defined as `payable`, but any ETH sent with the transaction is neither tracked nor transferred. This results in the ETH being stuck in the contract indefinitely.

Impact:

- ETH sent along with the transaction could be lost or stuck in the contract, leading to user dissatisfaction and possible financial losses.

Recommendation:

- Remove the `payable` modifier from the function or implement a mechanism to transfer incoming ETH to a designated address.

```
function buy(uint256 amountIn, address tokenAddress) public payable returns (bool)
{
    // Logic for purchasing asset token here

    // If ETH is sent, transfer it out to a defined address
    if (msg.value > 0) {
        payable(owner()).transfer(msg.value);  // Transfer ETH to owner or another
address
    }

    return true;
}
```

[L-03] Incorrect usage of `AccessControl` in upgradeable contracts

In the current contract, `AccessControl` (non-upgradeable version) is used in an upgradeable contract. Using the non-upgradeable variant of `AccessControl` can result in issues with uninitialized state, potential security vulnerabilities, and failure to follow upgradeability patterns. Specifically, upgradeable contracts should utilize `AccessControlUpgradeable` from OpenZeppelin to ensure proper initialization and compatibility with upgradeable contract patterns.

```
import "@openzeppelin/contracts-upgradeable/access/AccessControlUpgradeable.sol";
import "@openzeppelin/contracts-upgradeable/utils/PausableUpgradeable.sol"; import
"
```

[I-01] Use `Ownable2StepUpgradeable` for Ownership Transfers

Description:
To mitigate the risk of ownership issues, it's recommended to use the `Ownable2StepUpgradeable` contract for ownership transfers.

Impact:

- Prevents scenarios where ownership transfers could fail due to the next owner being non-existent or not accepting ownership.

Recommendation:

- Replace current ownership logic with `Ownable2StepUpgradeable` for better security.

```
import "@openzeppelin/contracts-upgradeable/access/Ownable2StepUpgradeable.sol";

contract Bonding is Ownable2StepUpgradeable {
    // Your contract logic here
}
```

[G-01] Validation of Pair Existence for Gas Optimization

Description:
When deriving the pair address in the `Bonding.sol` and `FRouter.sol` contracts, the existence of the pair address is not checked before performing operations, which can result in unnecessary gas consumption and failed transactions.

Impact:

- Increased gas costs and failed operations when the pair does not exist.

Recommendation:

- Perform a validation check to ensure the pair exists before proceeding with operations.

```
address pairAddress = factory.getPair(token, assetToken);
if (pairAddress == address(0)) {
    revert ZeroPairAddress();  // Revert if pair does not exist
}
```