#Bribes.sol

This contract is a bribe contract that allows users to deposit their tokens as a bribe to incentivize voters to vote in a specific way. The contract supports multiple reward tokens and distributes rewards over a period of 7 days. Below is a list of the contract's functions:

1. getEpochStart(): Returns the timestamp of the current epoch start.
2. getNextEpochStart(): Returns the timestamp of the next epoch start.
3. addReward(address \_rewardsToken): Allows the owner or the bribe factory to add a new reward token to the contract.
4. rewardsListLength(): Returns the number of reward tokens stored in the contract.
5. totalSupply(): Returns the total amount of tokens deposited in the current epoch.
6. totalSupplyAt(uint256 \_timestamp): Returns the total amount of tokens deposited at a specific timestamp.
7. balanceOfAt(uint256 tokenId, uint256 \_timestamp): Returns the balance of a specific token ID at a specific timestamp.
8. balanceOf(uint256 tokenId): Returns the balance of a specific token ID at the start of the next epoch.
9. earned(uint256 tokenId, address \_rewardToken): Returns the total amount of rewards earned by a specific token ID for a specific reward token.
10. rewardPerToken(address \_rewardsToken, uint256 \_timestmap): Returns the reward per token for a specific reward token at a specific timestamp.
11. \_deposit(uint256 amount, uint256 tokenId): Allows a user to deposit tokens as a bribe.
12. \_withdraw(uint256 amount, uint256 tokenId): Allows a user to withdraw their deposited tokens.
13. getReward(uint tokenId, address[] memory tokens): Allows a user to claim their earned rewards for a specific token ID and a list of reward tokens.
14. getRewardForOwner(uint tokenId, address[] memory tokens): Allows the contract owner to claim the earned rewards for a specific token ID and a list of reward tokens.
15. notifyRewardAmount(address \_rewardsToken, uint256 reward): Allows the owner or the bribe factory to notify the contract of a new reward amount for a specific reward token.
16. recoverERC20(address tokenAddress, uint256 tokenAmount): Allows the owner to recover accidentally sent ERC20 tokens.
17. setVoter(address \_Voter): Allows the owner to set a new Voter contract address.
18. setMinter(address \_minter): Allows the owner to set a new Minter contract address.
19. addRewardToken(address \_token): Allows the owner to add a new reward token to the contract.
20. setOwner(address \_owner): Allows the owner to set a new contract owner.
21. onlyOwner(): A modifier that restricts the function to be executed only by the contract owner.
22. nonReentrant(): A modifier that prevents a function from being reentrant.
23. RewardAdded: An event that is emitted when a new reward is added to the contract.
24. Staked: An event that is emitted when a user deposits tokens as a bribe.
25. Withdrawn: An event that is emitted when a user withdraws their deposited tokens.
26. RewardPaid: An event that is emitted when a user claims their earned rewards.
27. Recovered: An event that is emitted when the contract owner recovers ERC20 tokens.

#BribesDistribution.sol

This is a smart contract facilitates the distribution of LP (liquidity provider) fees to different bribe contracts. The contract has the following functions:

1. initialize(address \_voter): Initializes the contract with the Voter contract address.
2. distributeLPFees(): Distributes LP fees to the bribe contracts that are associated with the pairs in the pairs array. This function can only be called by the contract owner or addresses that have been added to the isAllowed mapping.
3. \_addPair(address \_pair): Internal function that adds a new pair to the pairs array.
4. addPair(address \_pair): Adds a new pair to the pairs array. This function can only be called by the contract owner.
5. addPairs(address[] memory \_pairs): Adds multiple pairs to the pairs array. This function can only be called by the contract owner.
6. removePair(address \_pair): Removes a pair from the pairs array. This function can only be called by the contract owner.
7. setVoter(address \_voter): Sets a new Voter contract address. This function can only be called by the contract owner.
8. setBribeOwner(address \_newOwner, address \_pair): Sets a new owner for the bribe contract associated with a specific pair. This function can only be called by the contract owner.
9. setAllowed(address \_allowed): Adds an address to the isAllowed mapping. This function can only be called by the contract owner.
10. removeAllowed(address \_allowed): Removes an address from the isAllowed mapping. This function can only be called by the contract owner.

The contract also includes several state variables, including:

1. pairs: An array of pair addresses that are associated with bribe contracts.
2. voter: The address of the Voter contract.
3. isPair: A mapping that tracks whether a pair has been added to the pairs array.
4. isAllowed: A mapping that tracks whether an address is allowed to call the distributeLPFees function.

#Echo.sol

This smart contract is an ERC20 token contract named "Echo" that implements the functions defined in the IEcho interface.

The contract has the following state variables:

* name: a string that represents the name of the token.
* symbol: a string that represents the symbol of the token.
* decimals: an unsigned integer that represents the number of decimals used in the token's display.
* totalSupply: an unsigned integer that represents the total supply of the token.
* balanceOf: a mapping that maps an address to its token balance.
* allowance: a mapping that maps an address to another address and the amount of tokens that the former has approved for the latter to spend.
* initialMinted: a boolean that indicates whether the initial minting has been done.
* minter: an address that represents the account that has the right to mint new tokens.
* redemptionReceiver: an address that represents the account that will receive tokens when they are redeemed.
* merkleClaim: an address that represents the account that is authorized to issue Merkle claims.

The contract has the following functions:

1. setMinter(address \_minter) external: Sets the minter address to \_minter. This function can only be called by the current minter.
2. initialMint(address \_recipient) external: Mints 200 million tokens and sends them to \_recipient. This function can only be called by the minter, and it can only be called once.
3. approve(address \_spender, uint \_value) external returns (bool): Approves the spender address \_spender to spend \_value amount of tokens from the caller's account.
4. \_mint(address \_to, uint \_amount) internal returns (bool): Mints \_amount number of tokens and sends them to \_to. This is an internal function that is used by other functions to mint new tokens.
5. \_transfer(address \_from, address \_to, uint \_value) internal returns (bool): Transfers \_value amount of tokens from \_from account to \_to account. This is an internal function that is used by other functions to transfer tokens.
6. transfer(address \_to, uint \_value) external returns (bool): Transfers \_value amount of tokens from the caller's account to \_to account.
7. transferFrom(address \_from, address \_to, uint \_value) external returns (bool): Transfers \_value amount of tokens from \_from account to \_to account. This function can only be called by an account that has been approved by \_from.
8. mint(address account, uint amount) external returns (bool): Mints \_amount number of tokens and sends them to account. This function can only be called by the minter.

#EchoGovernor.sol

This smart contract is a governance contract named "EchoGovernor" that inherits from several other governance contracts provided by OpenZeppelin. The contract implements the functions defined in the IGovernor interface.

The contract has the following state variables:

* team: an address that represents the team responsible for governing the contract.
* MAX\_PROPOSAL\_NUMERATOR: a uint256 that represents the maximum numerator value for the proposal threshold.
* PROPOSAL\_DENOMINATOR: a uint256 that represents the denominator value for the proposal threshold.
* proposalNumerator: a uint256 that represents the current numerator value for the proposal threshold.

The contract has the following functions:

1. votingDelay() public pure override(IGovernor) returns (uint256): Returns the delay between the time a proposal is made and when voting starts.
2. votingPeriod() public pure override(IGovernor) returns (uint256): Returns the period in which a proposal can be voted on.
3. setTeam(address newTeam) external: Sets the team address to newTeam. This function can only be called by the current team.
4. setProposalNumerator(uint256 numerator) external: Sets the proposal numerator to numerator. This function can only be called by the current team and the numerator must not exceed MAX\_PROPOSAL\_NUMERATOR.
5. proposalThreshold() public view override(L2Governor) returns (uint256): Returns the minimum number of votes required for a proposal to be accepted. The value is calculated based on the current proposalNumerator and the total supply of tokens at the current block timestamp.
6. constructor(IVotes \_ve): Initializes the contract with the name "Echo Governor", sets the team to the contract deployer, and sets the quorum fraction to 4%. It also initializes the contract with the \_ve Votes contract.
7. The contract also inherits several other governance contracts from OpenZeppelin, including L2Governor, L2GovernorCountingSimple, L2GovernorVotes, and L2GovernorVotesQuorumFraction.

#EchoHolders.sol

This smart contract is an ERC721 NFT (Non-Fungible Token) contract that is used to mint and manage NFTs. The contract extends the ERC721Enumerable contract from OpenZeppelin, which provides additional functionality for generating and managing NFTs.

The contract has the following state variables:

* \_baseURIextended: a string that represents the base URI for the tokens.
* MAX\_SUPPLY: a uint256 that represents the maximum number of tokens that can be minted.
* NFT\_PRICE: a uint256 that represents the price of each NFT in BNB.
* MAX\_PER\_MINT: a uint256 that represents the maximum number of NFTs that can be minted per transaction.
* SALE\_START\_TIMESTAMP: a uint256 that represents the timestamp when the sale starts.
* MAX\_RESERVE: a uint256 that represents the maximum amount of tokens that can be reserved.
* reservedAmount: a uint256 that represents the current number of reserved tokens.
* root: a bytes32 that represents the root hash of the Merkle tree used for whitelisting.
* multiSig: an address that represents the multi-sig wallet used for withdrawal of funds.
* firstMint: a mapping that maps addresses to boolean values indicating whether the address has already minted in the first round.
* secondMint: a mapping that maps addresses to uint256 values indicating the number of tokens the address has minted in the second round.
* originalMinters: a mapping that maps addresses to uint256 values indicating the total number of tokens the address has minted.

The contract has the following functions:

1. withdraw() external onlyOwner: Allows the contract owner to withdraw the funds from the contract to the multiSig wallet.
2. setRoot(bytes32 \_root) external onlyOwner: Sets the root variable to the provided \_root value.
3. setNftPrice(uint256 \_nftPrice) external onlyOwner: Sets the NFT\_PRICE variable to the provided \_nftPrice value.
4. reserveNFTs(address \_to, uint256 \_amount) external onlyOwner: Allows the contract owner to reserve \_amount tokens for \_to.
5. \_baseURI() internal view virtual override returns (string memory): Overrides the base URI function from ERC721Enumerable to return the \_baseURIextended variable.
6. baseURI() external view returns (string memory): Returns the base URI for the tokens.
7. setBaseURI(string memory baseURI\_) external onlyOwner: Sets the \_baseURIextended variable to the provided baseURI\_ value.
8. tokensOfOwner(address \_owner) external view returns (uint256[] memory): Returns an array of token IDs owned by the provided \_owner.
9. verifyLeaf(bytes32[] memory proof, address sender) internal view returns (bool): Verifies whether the sender's address is whitelisted using the Merkle proof provided in the proof argument.
10. mintFirst(bytes32[] memory proof) public payable: Allows users to mint NFTs in the first round of the sale. The user must be whitelisted and must send the correct amount of BNB. The maximum number of tokens that can be minted in the first round is 1 per user.
11. mintSecond(uint256 amount, bytes32[] memory proof) public payable: Allows users to mint NFTs in the second round of the sale. The user must be whitelisted and must send the correct amount of BNB. The maximum number of tokens that can be minted in the second round is 10 per user.
12. mintPublic(uint256 amount) public payable: Allows users to mint NFTs in the public sale. The user must send the correct amount of BNB and cannot mint more than 10 tokens at a time or own more than 15 tokens.
13. \_mintTo(address account, uint amount) internal: Internal function that mints the specified number of tokens to the specified account. The function ensures that the total number of tokens minted does not exceed the MAX\_SUPPLY.

#EchoLibrary.sol Decompiled !

This is a Solidity contract called EchoLibrary that implements several functions related to decentralized exchange (DEX) trading on the Ethereum blockchain, specifically involving the Echo V2 decentralized exchange.

The contract has the following state variables:

* router: an IRouter instance that represents the EchoSwap router contract.

The contract has the following functions:

1. constructor(address \_router): Initializes the router variable with the provided \_router address.
2. \_f(uint x0, uint y) internal pure returns (uint): Internal pure function that calculates a value based on two input parameters and returns the result.
3. \_d(uint x0, uint y) internal pure returns (uint): Internal pure function that calculates a value based on two input parameters and returns the result.
4. \_get\_y(uint x0, uint xy, uint y) internal pure returns (uint): Internal pure function that calculates a value based on three input parameters and returns the result.
5. getTradeDiff(uint amountIn, address tokenIn, address tokenOut, bool stable) external view returns (uint a, uint b): External view function that calculates the difference in trade price between two tokens on EchoSwap. The function takes in the input token amount, input token address, output token address, and a boolean flag indicating whether the output token is a stablecoin. The function returns two values - a represents the difference in price between the two tokens in terms of the output token, and b represents the difference in price between the two tokens in terms of the input token.
6. getTradeDiff(uint amountIn, address tokenIn, address pair) external view returns (uint a, uint b): External view function that calculates the difference in trade price between two tokens on EchoSwap. The function takes in the input token amount, input token address, and the address of a EchoSwap pair contract that contains the two tokens. The function returns two values - a represents the difference in price between the two tokens in terms of the output token, and b represents the difference in price between the two tokens in terms of the input token.
7. getSample(address tokenIn, address tokenOut, bool stable) external view returns (uint): External view function that calculates the price of a token pair on EchoSwap. The function takes in the input token address, output token address, and a boolean flag indicating whether the output token is a stablecoin. The function returns a single value representing the price of the token pair in terms of the output token.
8. getMinimumValue(address tokenIn, address tokenOut, bool stable) external view returns (uint, uint, uint): External view function that calculates the minimum value of a token pair on EchoSwap. The function takes in the input token address, output token address, and a boolean flag indicating whether the output token is a stablecoin. The function returns three values - sample represents the price of the token pair in terms of the output token, r0 represents the reserve of the input token in the EchoSwap pair contract, and r1 represents the reserve of the output token in the pair contract.
9. getAmountOut(uint amountIn, address tokenIn, address tokenOut, bool stable) external view returns (uint): External view function that calculates the output token amount for a given input token amount on EchoSwap. The function takes in the input token amount, input token address, output token address, and a boolean flag indicating whether the output token is a stablecoin. The function returns a single value representing the output token amount.
10. \_getAmountOut(uint amountIn, address tokenIn, uint \_reserve0, uint \_reserve1, address token0, uint decimals0, uint decimals1, bool stable) internal pure returns (uint): Internal pure function that calculates the output token amount for a given input token amount on EchoSwap. The function takes in the input token amount, input token address, reserve of the input token, reserve of the output token, address of the token that has a lower index in the EchoSwap pair contract, decimal values for the input and output tokens, and a boolean flag indicating whether the output token is a stablecoin. The function returns a single value representing the output token amount.
11. \_k(uint x, uint y, bool stable, uint decimals0, uint decimals1) internal pure returns (uint): Internal pure function that calculates a value based on four input parameters and returns the result. This function is used in the \_getAmountOut function.

#ExternalBribe.sol

This is a Solidity smart contract for an ExternalBribe that pays out rewards for a given pool based on the votes that were received from the user (goes hand in hand with Voter.vote()). The contract implements the IBribe interface. Here is a description of the functions and variables:

Variables:

1. voter: address of the voter contract that interacts with this contract.
2. \_ve: address of the VoterEscrow contract.
3. DURATION: a constant that represents the duration of the voting period in seconds.
4. MAX\_REWARD\_TOKENS: a constant that represents the maximum number of reward tokens that can be used.
5. PRECISION: a constant that represents the precision to use for calculations.
6. totalSupply: the total supply of tokens.
7. balanceOf: a mapping of token IDs to their balance.
8. tokenRewardsPerEpoch: a mapping of addresses to a mapping of epochs to the rewards per token for that epoch.
9. periodFinish: a mapping of addresses to the end time of the current voting period for that address.
10. lastEarn: a mapping of addresses to a mapping of epochs to the last time that the user earned rewards for that epoch.
11. rewards: an array of addresses for the reward tokens that can be used.
12. isReward: a mapping of reward token addresses to a boolean indicating whether that token is a valid reward.
13. Structs:
14. Checkpoint: a struct that represents a balance checkpoint for an account, including the timestamp and balance.
15. SupplyCheckpoint: a struct that represents a supply checkpoint for a token, including the timestamp and supply.

Structs:

* Checkpoint: a struct that represents a balance checkpoint for an account, including the timestamp and balance.
* SupplyCheckpoint: a struct that represents a supply checkpoint for a token, including the timestamp and supply.

Functions:

1. constructor: initializes the ExternalBribe contract with the address of the voter contract and an array of allowed reward tokens.
2. getEpochStart: given a timestamp, returns the start time of the current voting epoch.
3. getPriorBalanceIndex: given a token ID and a timestamp, returns the index of the balance checkpoint for that token at the given timestamp.
4. getPriorSupplyIndex: given a timestamp, returns the index of the supply checkpoint at the given timestamp.
5. \_writeCheckpoint: given a token ID and a balance, writes a new balance checkpoint for that token.
6. \_bribeStart: given a timestamp, returns the start time of the bribe period for that timestamp.
7. \_unlocked: an internal variable used to prevent reentrancy.
8. lock: a modifier that prevents reentrancy.
9. notifyRewardAmount: a function that updates the reward amount for a given token and epoch.
10. withdraw: a function that allows a user to withdraw tokens from their balance.
11. deposit: a function that allows a user to deposit tokens into their balance.
12. claimRewards: a function that allows a user to claim their rewards for a given epoch and reward token.
13. \_writeSupplyCheckpoint(): an internal function that writes a new supply checkpoint. It does not take any inputs and does not return anything.
14. rewardsListLength(): an external function that returns the length of the rewards array. It takes no inputs and returns a uint.
15. lastTimeRewardApplicable(address token): a public function that returns the last time a reward was modified or the end of the current voting period for a given token. It takes an address token as input and returns a uint.
16. getReward(uint tokenId, address[] memory tokens): an external function that allows a user to claim rewards for a given token. It takes a uint tokenId and an array of reward token addresses tokens as inputs and does not return anything.
17. getRewardForOwner(uint tokenId, address[] memory tokens): an external function used by the voter contract to allow batched reward claims. It takes a uint tokenId and an array of reward token addresses tokens as inputs and does not return anything.
18. earned(address token, uint tokenId): a public function that returns the amount of rewards earned for a given token and epoch. It takes an address token and a uint tokenId as inputs and returns a uint.
19. \_deposit(uint amount, uint tokenId): an external function used internally to deposit tokens into the balance of a given token ID. It takes a uint amount and a uint tokenId as inputs and does not return anything.
20. \_withdraw(uint amount, uint tokenId): an external function used internally to withdraw tokens from the balance of a given token ID. It takes a uint amount and a uint tokenId as inputs and does not return anything.
21. left(address token): an external function that returns the remaining rewards for a given token. It takes an address token as input and returns a uint.
22. notifyRewardAmount(address token, uint amount): an external function that updates the reward amount for a given token and epoch. It takes an address token and a uint amount as inputs and does not return anything.
23. swapOutRewardToken(uint i, address oldToken, address newToken): an external function that swaps out an old reward token for a new one. It takes a uint i, an address oldToken, and an address newToken as inputs and does not return anything.
24. \_safeTransfer(address token, address to, uint256 value): an internal function used to safely transfer tokens. It takes an address token, an address to, and a uint256 value as inputs and does not return anything.
25. \_safeTransferFrom(address token, address from, address to, uint256 value): an internal function used to safely transfer tokens from one address to another. It takes an address token, an address from, an address to, and a uint256 value as inputs and does not return anything.

#Gauge.sol

This is a Solidity contract called Gauge, which implements the IGauge interface. The contract is used to incentivize pools and emit reward tokens over 7 days for staked LP tokens.

The contract also includes several mappings and structs to keep track of user balances, reward rates, supply checkpoints, and reward-per-token checkpoints, as well as various events to log transactions.

Here is a summary of its main functions:

1. Constructor: it initializes the contract by setting several variables, including the address of the LP token that needs to be staked for rewards, the address of the ve token used for gauges, the addresses of the internal and external bribe contracts, the address of the voter contract, a boolean flag indicating whether the gauge is for a pair, and an array of allowed reward tokens.
2. claimFees(): it claims any fees earned from the LP token and distributes them to the internal and external bribe contracts based on their remaining allowance.
3. getPriorBalanceIndex(): it determines the prior balance for an account as of a block number.
4. \_claimFees(): an internal function that claims fees earned by the LP token and distributes them to the bribe contracts.
5. getPriorSupplyIndex(uint timestamp): This function takes a timestamp as input and returns an index indicating the prior balance of the supply as of the specified timestamp.
6. getPriorRewardPerToken(address token, uint timestamp): This function takes a token address and a timestamp as inputs and returns the reward per token for the specified token as of the specified timestamp.
7. \_writeCheckpoint(address account, uint balance): This internal function takes an account address and a balance as inputs and writes a checkpoint for the account.
8. \_writeRewardPerTokenCheckpoint(address token, uint reward, uint timestamp): This internal function takes a token address, a reward amount, and a timestamp as inputs and writes a reward per token checkpoint for the specified token.
9. \_writeSupplyCheckpoint(): This internal function writes a checkpoint for the supply.
10. rewardsListLength(): This external function returns the length of the rewards array.
11. lastTimeRewardApplicable(address token): This function takes a token address as input and returns the last time the reward was modified or the period finish time if the reward has ended.
12. getReward(address account, address[] memory tokens): This function takes an account address and an array of token addresses as inputs and distributes rewards to the account for each specified token.
13. rewardPerToken(address token): This function takes a token address as input and returns the reward per token for the specified token.
14. derivedBalance(address account): This function takes an account address as input and returns the derived balance for the specified account.
15. batchRewardPerToken(address token, uint maxRuns): This external function takes a token address and a maximum number of runs as inputs and updates the stored reward per token values for the specified token without the last one snapshot.
16. \_batchRewardPerToken(address token, uint maxRuns): This internal function takes a token address and a maximum number of runs as inputs and updates the stored reward per token values for the specified token without the last one snapshot.
17. \_calcRewardPerToken(address token, uint timestamp1, uint timestamp0, uint supply, uint startTimestamp): This internal function takes a token address, two timestamps, a supply amount, and a start timestamp as inputs and calculates the reward per token for the specified token.
18. \_updateReward(address account, address token): This internal function takes an account address and a token address as inputs and updates the reward earned and the last update time for the specified account and token.
19. updateReward(address account, address[] memory tokens): This external function takes an account address and an array of token addresses as inputs and updates the rewards earned for the specified account and tokens.
20. notifySupplyUpdated(): This external function notifies the contract that the supply has been updated and updates the supply checkpoint.
21. notifyRewardAmount(address token, uint reward): This external function notifies the contract that a reward has been added for the specified token and updates the reward per token checkpoint.
22. \_updateRewardPerToken(address token, uint maxRuns, bool actualLast) internal returns (uint, uint): This function updates the reward per token value for a given token. It takes three inputs: the token address, the maximum number of runs to perform, and a boolean indicating whether to use the actual last value or the last value from the previous action. It returns a tuple containing the updated reward per token value and the start timestamp.
23. earned(address token, address account) public view returns (uint): This function calculates the estimated amount of rewards earned by a given account for a given token. It takes two inputs: the token address and the account address. It returns the estimated earned rewards.
24. depositAll(uint tokenId) external: This function deposits the entire balance of the stake token for the calling user. It takes one input: the ID of the voting escrow token to attach to the deposited funds.
25. deposit(uint amount, uint tokenId) public lock: This function deposits a specified amount of funds for the calling user. It takes two inputs: the amount to deposit and the ID of the voting escrow token to attach to the deposited funds.
26. withdrawAll() external: This function withdraws the entire balance of funds for the calling user.
27. withdraw(uint amount) public: This function withdraws a specified amount of funds for the calling user.
28. withdrawToken(uint amount, uint tokenId) public lock: This function withdraws a specified amount of funds for the calling user, along with detaching a specified voting escrow token from the funds.
29. left(address token) external view returns (uint): This function returns the remaining reward amount for a given token.
30. notifyRewardAmount(address token, uint amount) external lock: This function notifies the contract of a new reward amount for a given token. It takes two inputs: the token address and the new reward amount.
31. swapOutRewardToken(uint i, address oldToken, address newToken) external: This function allows the contract owner to swap out an old reward token for a new one. It takes three inputs: the index of the old token, the address of the old token, and the address of the new token.
32. \_safeTransfer(address token, address to, uint256 value) internal: This internal function safely transfers a specified amount of tokens to a given address. It takes three inputs: the token address, the recipient address, and the amount to transfer.
33. \_safeTransferFrom(address token, address from, address to, uint256 value) internal: This internal function safely transfers a specified amount of tokens from a given address to a specified recipient address. It takes four inputs: the token address, the sending address, the recipient address, and the
34. \_safeApprove(address token, address spender, uint256 value) internal: This internal function safely approves a specified amount of tokens to be spent by a given spender address. It takes three inputs: the token address, the spender address, and the amount to approve.

#InetrnalBribe.sol

This contract is an implementation of a bribe scheme that pays out rewards for a given pool based on the votes that were received from the user. It is written in Solidity and has a version of 0.8.13.

The contract defines an interface for a bribe contract, which includes the following functions:

1. getReward(uint tokenId, address[] memory tokens) external lock: This function allows a user to claim rewards for a given token. It takes in the token ID and an array of reward tokens as inputs. It then updates the reward per token stored for each reward token, calculates the amount of rewards earned by the user for each reward token, and transfers the rewards to the user. It emits a ClaimRewards event for each reward token.
2. getRewardForOwner(uint tokenId, address[] memory tokens) external lock: This function is similar to getReward, but is intended to be called by the owner of the token (i.e. the Voter contract). It updates the reward per token stored and calculates the earned rewards for each reward token, but transfers the rewards to the owner of the token instead of the user. It emits a ClaimRewards event for each reward token.
3. \_updateRewardPerToken(address token, uint timestamp, bool distribute) internal returns (uint, uint): This function updates the reward per token stored for a given token and returns the updated value and the timestamp of the update. It takes in the token address, a timestamp, and a boolean flag indicating whether or not to distribute rewards. If the distribute flag is true, the function also calculates and distributes rewards to the users based on the updated reward per token. This function is used by getReward and getRewardForOwner to update the reward per token stored before calculating earned rewards.
4. \_safeTransfer(address token, address to, uint amount) internal: This function transfers a specified amount of tokens to a specified address. It takes in the token address, the recipient address, and the amount of tokens to transfer. It first checks if the token balance of the contract is sufficient to cover the transfer, and reverts if it is not. It then transfers the tokens to the recipient address.
5. rewardPerToken(address token): This function returns the reward per token for a specific token. It takes a token address as an input and returns a uint value.
6. batchRewardPerToken(address token, uint maxRuns): This function updates the reward per token for a specific token based on multiple checkpoints. It takes a token address and maximum number of runs as inputs, and updates the rewardPerTokenStored and lastUpdateTime values for the token.
7. \_batchRewardPerToken(address token, uint maxRuns): This is an internal function that is called by batchRewardPerToken. It updates the reward per token for a specific token based on multiple checkpoints and returns the updated rewardPerTokenStored and lastUpdateTime values.
8. \_calcRewardPerToken(address token, uint timestamp1, uint timestamp0, uint supply, uint startTimestamp): This is an internal function that calculates the reward per token for a specific token based on two timestamps, the supply, and the startTimestamp.
9. batchUpdateRewardPerToken(address token, uint maxRuns): This function updates the rewardPerTokenStored and lastUpdateTime values for all tokens based on multiple checkpoints. It takes a token address and maximum number of runs as inputs and updates the values for all tokens.
10. \_updateRewardForAllTokens(): This is an internal function that is called by batchUpdateRewardPerToken. It updates the rewardPerTokenStored and lastUpdateTime values for all tokens based on multiple checkpoints.
11. earned(address token, uint tokenId): This function calculates the amount of reward earned by a user for a specific token. It takes a token address and tokenId as inputs and returns a uint value.
12. \_deposit(uint amount, uint tokenId): This is an internal function that is called when a user deposits tokens into the liquidity pool. It updates the totalSupply and balanceOf values for the token and writes a checkpoint.
13. \_withdraw(uint amount, uint tokenId): This is an internal function that is called when a user withdraws tokens from the liquidity pool. It updates the totalSupply and balanceOf values for the token and writes a checkpoint.
14. left(address token): This function returns the amount of reward remaining for a specific token. It takes a token address as an input and returns a uint value.
15. notifyRewardAmount(address token, uint amount): This function notifies the contract of the amount of reward to be distributed for a specific token. It takes a token address and amount as inputs and updates the rewardRate, periodFinish, and rewardPerTokenStored values for the token.
16. swapOutRewardToken(uint i, address oldToken, address newToken): This function swaps out an old reward token for a new one. It takes the index of the old token, the old token address, and the new token address as inputs and updates the rewards array and isReward mapping.
17. \_safeTransfer(address token, address to, uint256 value): This is an internal function that is used to transfer tokens safely.
18. \_safeTransferFrom(address token, address from, address to, uint256 value): This is an internal function that is used to transfer tokens safely from a specific address.

#Minter.sol

This contract is a token minter contract that allows for minting new tokens and distributing them to various parties according to specific rules. The contract contains the following functions:

Variables:

* isFirstMint: a boolean flag indicating whether or not this is the first time minting tokens
* EMISSION: an integer representing the emission rate for the token
* TAIL\_EMISSION: an integer representing the tail emission rate for the token
* REBASEMAX: an integer representing the maximum rebase rate for the token
* PRECISION: an integer representing the precision for the token
* teamRate: an integer representing the rate at which tokens are distributed to the team
* MAX\_TEAM\_RATE: an integer representing the maximum team rate for the token
* weekly: an integer representing the weekly supply of tokens to be minted
* active\_period: an integer representing the current active period for minting tokens
* initializer: the address of the initializer of the contract
* team: the address of the team that receives token distributions
* pendingTeam: the address of the pending team that may receive token distributions
* \_echo: an interface to the Echo token contract
* \_voter: an interface to the Voter contract
* \_ve: an interface to the VotingEscrow contract
* \_rewards\_distributor: an interface to the RewardsDistributor contract

Functions:

1. constructor: initializes the contract with the addresses of the Voter, VotingEscrow, and RewardsDistributor contracts
2. initialize: initializes the minting process by creating locks for specified claimants and transferring tokens to the VotingEscrow contract
3. setTeam: sets the pending team address
4. acceptTeam: accepts the pending team address as the new team address
5. setVoter: sets the Voter contract address
6. setTeamRate: sets the team rate for token distributions
7. setEmission: sets the emission rate for the token
8. setRebase: sets the rebase rate for the token
9. circulating\_supply: calculates and returns the circulating supply of tokens
10. calculate\_emission: calculates and returns the weekly emission rate for the token
11. weekly\_emission: calculates and returns the maximum weekly emission rate for the token
12. circulating\_emission: calculates and returns the circulating tail end emission rate for the token
13. calculate\_rebase: calculates and returns the rebase rate for the token
14. update\_period: updates the period for minting tokens and distributes tokens to various parties
15. check: checks whether the contract can update the period
16. period: returns the current period for minting tokens

#NFTRedeem.sol

This contract is a smart contract for redeeming NFTs for a specified amount of a specific ERC20 token. Here is a brief description of each function:

1. constructor: initializes the contract with the WBNB address, the NFT address, and the redeem amount
2. redeem: allows users to redeem their NFTs for the specified ERC20 token. It takes an array of token IDs and the address to send the tokens to as inputs
3. startRedeem: allows the contract owner to start the redemption process
4. stopRedeem: allows the contract owner to stop the redemption process
5. setRedeemAmount: allows the contract owner to set the redeem amount
6. depositRedeemAmount: allows users to deposit the specified ERC20 token into the contract
7. withdrawERC20: allows the contract owner to withdraw an ERC20 token from the contract
8. withdrawERC721: allows the contract owner to withdraw NFTs from the contract
9. \_safeTransfer: a helper function to safely transfer ERC20 tokens
10. \_safeTransferFrom: a helper function to safely transfer ERC20 tokens from the specified address

Variables:

* redeemActive: a boolean flag indicating whether or not the redemption process is active
* WBNB: the address of the WBNB contract
* NFT: the address of the NFT contract
* redeemAmount: the amount of the specified ERC20 token required for each NFT

#NFTSalesSplitter.sol

This contract is a smart contract for redeeming NFTs for a specified amount of a specific ERC20 token. Here is a brief description of each function:

1. constructor: initializes the contract with the WBNB address, the NFT address, and the redeem amount
2. redeem: allows users to redeem their NFTs for the specified ERC20 token. It takes an array of token IDs and the address to send the tokens to as inputs
3. startRedeem: allows the contract owner to start the redemption process
4. stopRedeem: allows the contract owner to stop the redemption process
5. setRedeemAmount: allows the contract owner to set the redeem amount
6. depositRedeemAmount: allows users to deposit the specified ERC20 token into the contract
7. withdrawERC20: allows the contract owner to withdraw an ERC20 token from the contract
8. withdrawERC721: allows the contract owner to withdraw NFTs from the contract
9. \_safeTransfer: a helper function to safely transfer ERC20 tokens
10. \_safeTransferFrom: a helper function to safely transfer ERC20 tokens from the specified address

Variables:

* redeemActive: a boolean flag indicating whether or not the redemption process is active
* WBNB: the address of the WBNB contract
* NFT: the address of the NFT contract
* redeemAmount: the amount of the specified ERC20 token required for each NFT

#Pair.sol

This is a Solidity smart contract for a decentralized exchange (DEX) Pair, which represents a pair of two ERC20 tokens on the blockchain. Here are the details of the contract:

Variables:

* name: a string variable representing the name of the pair
* symbol: a string variable representing the symbol of the pair
* decimals: an unsigned integer variable representing the number of decimal places used by the pair's tokens
* stable: a boolean variable indicating whether the pair is stable or volatile
* totalSupply: an unsigned integer variable representing the total supply of the pair's tokens
* allowance: a mapping that keeps track of the amount of tokens that a third party is allowed to spend on behalf of another address
* balanceOf: a mapping that keeps track of the balance of tokens owned by each address
* DOMAIN\_SEPARATOR: a bytes32 variable used in the permit function for permit approvals
* PERMIT\_TYPEHASH: a bytes32 variable used in the permit function for permit approvals
* nonces: a mapping that keeps track of the number of permit approvals made by an address
* MINIMUM\_LIQUIDITY: a constant unsigned integer representing the minimum liquidity required for the pair
* token0: the address of the first token in the pair
* token1: the address of the second token in the pair
* fees: the address of the PairFees contract that handles fees for the pair
* factory: the address of the PairFactory contract that created the pair
* periodSize: a constant unsigned integer representing the length of a time period for local oracles
* observations: an array of Observation struct instances, representing the observations made by local oracles
* decimals0: an unsigned integer representing the number of decimal places used by the first token in the pair
* decimals1: an unsigned integer representing the number of decimal places used by the second token in the pair
* reserve0: an unsigned integer representing the reserve of the first token in the pair
* reserve1: an unsigned integer representing the reserve of the second token in the pair
* blockTimestampLast: an unsigned integer representing the timestamp of the last block
* reserve0CumulativeLast: an unsigned integer representing the cumulative reserve of the first token in the pair
* reserve1CumulativeLast: an unsigned integer representing the cumulative reserve of the second token in the pair
* index0: an unsigned integer representing the accumulated fee of the first token in the pair
* index1: an unsigned integer representing the accumulated fee of the second token in the pair
* supplyIndex0: a mapping that keeps track of the index0 value for each LP
* supplyIndex1: a mapping that keeps track of the index1 value for each LP
* claimable0: a mapping that keeps track of the amount of unclaimed fees for the first token in the pair
* claimable1: a mapping that keeps track of the amount of unclaimed fees for the second token in the pair

Functions:

1. observationLength(): returns the length of the observations array
2. lastObservation(): returns the last observation made by the local oracle
3. metadata(): returns metadata about the pair- swap(uint amount0Out, uint amount1Out, address to, bytes calldata data): allows a user to swap tokens in the pair. The function takes in the desired output amounts of each token, the address to send the swapped tokens to, and optional data. The function first checks if the output amounts are non-zero and valid, then calculates the input amounts based on the current reserves and the desired output amounts. The function then transfers the input tokens from the sender to the pair contract, swaps the tokens using the swapInternal function, and transfers the output tokens to the designated address.
4. skim(address to): allows the fees contract to skim the excess tokens from the pair. The function calculates the excess tokens based on the current reserves and the last observation, then transfers the excess tokens to the fees contract.
5. sync(): allows the pair to update its reserves and the last observation made by the local oracle. The function first checks if the time period for the local oracle has passed, then updates the reserves and the last observation based on the current block timestamp and the current balances of the tokens in the pair.
6. initialize(uint \_balance0, uint \_balance1): allows the pair to be initialized with initial balances of each token. The function can only be called once by the factory that created the pair, and sets the initial reserves and total supply of the LP tokens.
7. mint(address to): allows a user to mint LP tokens for the pair. The function calculates the amount of LP tokens to mint based on the current reserves and the amount of tokens sent by the user, then transfers the tokens to the pair contract and mints the corresponding amount of LP tokens to the designated address.
8. burn(address to): allows a user to burn LP tokens for the pair. The function calculates the amount of tokens to send to the user based on the current reserves and the amount of LP tokens burned, then transfers the tokens to the designated address and burns the corresponding amount of LP tokens.
9. permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, bytes32 s): allows a user to approve a third party to spend their tokens on their behalf using EIP-712 permit approvals. The function first checks if the permit has expired, then verifies the signature using the DOMAIN\_SEPARATOR, PERMIT\_TYPEHASH, and nonces variables. The function then sets the allowance for the designated address.
10. transfer(address to, uint value): allows a user to transfer their tokens to another address. The function first checks if the transfer is valid, then transfers the tokens from the sender to the designated address.
11. transferFrom(address from, address to, uint value): allows a third party with approval to spend tokens on behalf of another address to transfer tokens to another address. The function first checks if the transfer is valid and if the third party has approval to spend the tokens, then transfers the tokens from the sender to the designated address and updates the allowance.
12. \_update0(uint amount): This is an internal function that updates the fees accrued on token0. It takes an input parameter amount, which represents the amount of token0 that was traded. The function calculates the referral fee based on the MAX\_REFERRAL\_FEE set by the PairFactory contract and transfers it to the dibs address. It also calculates the staking fee based on the stakingNFTFee set by the PairFactory contract and calls the processStakingFees function of the PairFees contract to process the fee. Finally, it removes the staking fee from the total amount and calculates the ratio of the remaining amount to the total supply of the LP tokens. If the ratio is greater than 0, it adds the ratio to index0. The function emits a Fees event with the total fees accrued.
13. \_update1(uint amount): This is an internal function that updates the fees accrued on token1. It takes an input parameter amount, which represents the amount of token1 that was traded. The function is similar to \_update0, but it calculates the fees for token1 instead of token0.
14. \_updateFor(address recipient): This is an internal function that updates the accrued fees for an LP token holder. It takes an input parameter recipient, which represents the address of the LP token holder. The function calculates the difference between the last adjusted index for the LP token holder and the current global index for both token0 and token1. If the LP token holder has a positive balance, the function adds the accrued difference for each token to the claimable0 and claimable1 variables. The function updates the LP token holder's current position to the global position and emits a Fees event with the total fees accrued.
15. getReserves() public view returns (uint \_reserve0, uint \_reserve1, uint \_blockTimestampLast): This is a public function that returns the current reserves of the token pair and the timestamp of the last block in which the reserves were updated. The function returns \_reserve0, \_reserve1, and \_blockTimestampLast.
16. \_update(uint balance0, uint balance1, uint \_reserve0, uint \_reserve1) internal: This is an internal function that updates the reserves of the token pair. It takes four input parameters: balance0 and balance1, which represent the current balances of token0 and token1, respectively, and \_reserve0 and \_reserve1, which represent the current reserves of token0 and token1, respectively. The function calculates the time elapsed since the last update and updates the cumulative reserves if necessary. It also records the current reserves and timestamp and emits a Sync event with the updated reserves.
17. currentCumulativePrices() public view returns (uint reserve0Cumulative, uint reserve1Cumulative, uint blockTimestamp): This is a public function that returns the current cumulative reserves of the token pair and the current block timestamp. If the last update on the pair was more than one block ago, it mocks the accumulated price values.
18. current(address tokenIn, uint amountIn) external view returns (uint amountOut): This is a public function that calculates the current price of tokenIn in terms of the other token. It takes two input parameters: tokenIn, which represents the token to be traded, and amountIn, which represents the amount of tokenIn to be traded. The function uses the current cumulative reserves and the last observation to calculate the amount of the other token that can be obtained for the given amount of tokenIn. It returns amountOut, which represents the amount of the other token that can be obtained.
19. quote(address tokenIn, uint amountIn, uint granularity) external view returns (uint amountOut): This function calculates the average price of tokenIn in terms of the other token over a window of time. It takes three input parameters: tokenIn, which represents the token to be traded, amountIn, which represents the amount of tokenIn to be traded, and granularity, which represents the number of observations to be included in the window. The function calls the sample function to get an array of token prices and calculates the average of these prices. It returns amountOut, which represents the average price of tokenIn.
20. prices(address tokenIn, uint amountIn, uint points) external view returns (uint[] memory): This function returns an array of token prices sampled at equally spaced intervals over a window of time. It takes three input parameters: tokenIn, which represents the token to be traded, amountIn, which represents the amount of tokenIn to be traded, and points, which represents the number of observations to be included in the array. The function calls the sample function to get an array of token prices and returns this array.
21. sample(address tokenIn, uint amountIn, uint points, uint window) public view returns (uint[] memory): This function returns an array of token prices sampled at equally spaced intervals over a window of time. It takes four input parameters: tokenIn, which represents the token to be traded, amountIn, which represents the amount of tokenIn to be traded, points, which represents the number of observations to be included in the array, and window, which represents the number of observations to be included in each window. The function calculates the prices for each observation in the array by iterating over the observation history and calling the \_getAmountOut function to get the price of tokenIn at each observation. It returns an array of uint values representing the prices of tokenIn.
22. mint(address to) external lock returns (uint liquidity): This function mints new liquidity tokens when users deposit tokens into the liquidity pool. It takes one input parameter to, which represents the address of the user who is depositing tokens. The function gets the current balances of the tokens in the liquidity pool and calculates the amount of liquidity tokens to mint based on the current total supply of liquidity tokens and the amount of tokens being deposited. It then mints the new liquidity tokens, updates the reserves of the tokens in the liquidity pool, and emits a Mint event.
23. burn(address to) external lock returns (uint amount0, uint amount1): This function burns liquidity tokens when users withdraw tokens from the liquidity pool. It takes one input parameter to, which represents the address of the user who is withdrawing tokens. The function gets the current balances of the tokens in the liquidity pool and calculates the amount of tokens to return to the user based on their share of the liquidity pool. It then transfers the tokens to the user, updates the reserves of the tokens in the liquidity pool, and emits a Burn event.
24. swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external lock: This function allows a user to swap tokens in the pool. The function takes in the following inputs:
    * amount0Out: The amount of token 0 that the user wants to receive in the swap.
    * amount1Out: The amount of token 1 that the user wants to receive in the swap.
    * to: The address to which the swapped tokens will be transferred.
    * data: Additional data that can be passed to the function.
    * msg.sender: The address of the user who initiated the swap.
    * amount0In: The amount of token 0 that the user provided in the swap.
    * amount1In: The amount of token 1 that the user provided in the swap.
    * amount0Out: The amount of token 0 that the user received in the swap.
    * amount1Out: The amount of token 1 that the user received in the swap.
    * to: The address to which the swapped tokens were transferred.
25. skim(address to) external lock: This function allows the contract owner to take any excess tokens that are left in the contract after a swap. The function takes in the following input: to: The address to which the excess tokens will be transferred. The function returns no value.
26. sync() external lock: This function ensures that the reserves of the tokens in the pool match the actual balances of the tokens held by the contract. The function takes no inputs and returns no value.
27. \_f(uint x0, uint y) internal pure returns (uint): This internal function calculates the value of a function f(x,y) used in calculating the amount of token 1 to receive in a swap. The function takes in the following inputs:
    * x0: The balance of token 0 in the pool.
    * y: The balance of token 1 in the pool.
    * The function returns a uint value.
28. \_get\_y(uint x0, uint xy, uint y) internal pure returns (uint): This internal function calculates the amount of token 1 to receive in a swap based on the input amount of token 0. The function takes in the following inputs:
    * x0: The balance of token 0 in the pool.
    * xy: The amount of token 0 provided in the swap.
    * y: The balance of token 1 in the pool.
    * The function returns a uint value.
29. getAmountOut(uint amountIn, address tokenIn) external view returns (uint): This function calculates the expected amount of output token that a user will receive for a given input amount of a token in the pool. The function takes in the following inputs:
    * amountIn: The input amount of token in the pool.
    * tokenIn: The address of the input token.
    * The function returns a uint value, which represents the expected amount of output token that the user will receive.
30. \_getAmountOut(uint amountIn, address tokenIn, uint \_reserve0, uint \_reserve1) internal view returns (uint): This internal function is used to calculate the expected amount of output token that a user will receive for a given input amount of a token in the pool. The function takes in the following inputs:
    * amountIn: The input amount of token in the pool.
    * tokenIn: The address of the input token.
    * \_reserve0: The balance of token 0 in the pool.
    * \_reserve1: The balance of token 1 in the pool.
    * The function returns a uint value, which represents the expected amount of output token that the user will receive.
31. \_k(uint x, uint y) internal view returns (uint): This internal function is used to calculate the constant value of the product of the reserve balances of the tokens in the pool. The function takes in the following inputs:
    * x: The balance of token 0 in the pool.
    * y: The balance of token 1 in the pool.
    * The function returns a uint value.
32. \_mint(address dst, uint amount) internal: This internal function is used to mint new tokens and add them to the balance of a user. The function takes in the following inputs:
    * dst: The address of the user who will receive the new tokens.
    * amount: The amount of tokens to mint.
    * The function returns no value.
33. \_burn(address dst, uint amount) internal: This internal function is used to burn tokens and subtract them from the balance of a user. The function takes in the following inputs:
    * dst: The address of the user who will lose the tokens.
    * amount: The amount of tokens to burn.
    * The function returns no value.
34. approve(address spender, uint amount) external returns (bool): This function is used to approve a spender to spend a certain amount of tokens on behalf of the user. The function takes in the following inputs:
    * spender: The address of the spender.
    * amount: The amount of tokens to approve.
    * The function returns a bool value indicating whether the approval was successful.
35. permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, bytes32 s) external: This function is used to approve a spender to spend a certain amount of tokens on behalf of the user using an EIP-712 signature. The function takes in the following inputs:
    * owner: The address of the token owner.
    * spender: The address of the spender.
    * value: The amount of tokens to approve.
    * deadline: The deadline for the approval to be used.
    * v, r, s: The EIP-712 signature parameters.
    * The function returns no value.
36. transfer(address dst, uint amount) external returns (bool): This function is used to transfer tokens from the sender to another user. The function takes in the following inputs:
    * dst: The address of the user who will receive the tokens.
    * amount: The amount of tokens to transfer.
    * The function returns a bool value indicating whether the transfer was successful.
37. transferFrom(address src, address dst, uint amount) external returns (bool): This function is used to transfer tokens from a specified sender to another user on behalf of the sender. The function takes in the following inputs:
    * src: The address of the token sender.
    * dst: The address of the user who will receive the tokens.
    * amount: The amount of tokens to transfer.
    * The function returns a bool value indicating whether the transfer was successful.
38. \_transferTokens(address src, address dst, uint amount) internal: This internal function is used to transfer tokens from one user to another. The function takes in the following inputs:
    * src: The address of the user who will send the tokens.
    * dst: The address of the user who will receive the tokens.
    * amount: The amount of tokens to transfer.
    * The function returns no value.
39. \_safeTransfer(address token, address to, uint256 value) internal: This internal function is used to safely transfer tokens from the contract to an external address. The function takes in the following inputs:
    * token: The address of the token to be transferred.
    * to: The address of the recipient.
    * value: The amount of tokens to transfer.
    * The function returnsno value.
40. \_safeApprove(address token, address spender, uint256 value) internal: This internal function is used to safely approve a spender to spend a certain amount of tokens on behalf of the contract. The function takes in the following inputs:
    * token: The address of the token to be approved.
    * spender: The address of the spender.
    * value: The amount of tokens to approve.

#PairFees.sol

This contract is a Pair Fees contract that is used to split fees between two tokens in a decentralized exchange (DEX) pair. It contains the following functions:

1. constructor(address \_token0, address \_token1): The constructor function is used to initialize the contract with the addresses of the two tokens in the pair. It takes in the following inputs:
   * \_token0: The address of the first token in the pair.
   * \_token1: The address of the second token in the pair.

The function sets the pair, token0, and token1 variables to the contract's address and the addresses of the two tokens in the pair, respectively.

1. \_safeTransfer(address token,address to,uint256 value) internal: This internal function is used to safely transfer tokens from the contract to an external address. The function takes in the following inputs:
   * token: The address of the token to be transferred.
   * to: The address of the recipient.
   * value: The amount of tokens to transfer.

The function checks that the token has a non-zero code length, then calls the transfer function of the token contract to transfer the tokens to the recipient. It then checks that the transfer was successful and reverts the transaction if it was not.

1. claimFeesFor(address recipient, uint amount0, uint amount1) external: This external function is used to allow the pair to transfer fees to users. The function takes in the following inputs:
   * recipient: The address of the user who will receive the fees.
   * amount0: The amount of token 0 fees to transfer.
   * amount1: The amount of token 1 fees to transfer.

The function requires that the caller is the pair contract and transfers the specified amounts of fees to the recipient using the \_safeTransfer function.

1. processStakingFees(uint amount, bool isTokenZero) external: This external function is used to process staking fees for the two tokens in the pair. The function takes in the following inputs:  
    amount: The amount of staking fees to process.  
    isTokenZero: A boolean value indicating whether the staking fees are for token 0 or token 1.  
   The function requires that the caller is the pair contract and adds the specified amount of staking fees to the appropriate variable (toStake0 or toStake1).
2. withdrawStakingFees(address recipient) external: This external function is used to withdraw staking fees for the two tokens in the pair. The function takes in the following input:  
   recipient: The address of the user who will receive the staking fees.  
   The function requires that the caller is the pair contract and transfers the staking fees from the appropriate variable (toStake0 or toStake1) to the recipient using the \_safeTransfer function. It then sets the variable to zero.

#PancakeMigrator.sol

This contract is a PancakeMigrator contract, which is used to migrate liquidity from Uniswap V2 LP tokens to PancakeSwap V2 LP tokens. It contains the following functions:

1. constructor(address \_pairFactory, address \_thRouter): The constructor function is used to initialize the contract with the addresses of the pair factory and the PancakeSwap router. It takes in the following inputs: \_pairFactory: The address of the Uniswap V2 pair factory.  
    \_thRouter: The address of the Uniswap V2 router.  
    The function sets the pairFactory and thRouter variables to the corresponding interfaces of the Uniswap V2 pair factory and router.
2. migrate(address \_lp, uint \_amount, bool stable) external nonReentrant: This external function is used to migrate Uniswap V2 LP tokens to PancakeSwap V2 LP tokens. The function takes in the following inputs:   
   \_lp: The address of the Uniswap V2 LP token.  
    \_amount: The amount of Uniswap V2 LP tokens to migrate.  
    stable: A boolean value indicating whether to create or add to a stable or non-stable pair. true means the pair is stable.  
    The function first checks that the caller has enough Uniswap V2 LP tokens to migrate. It then removes liquidity from the Uniswap V2 LP token using the \_removeLiquidity internal function. After removing liquidity, it adds liquidity to the corresponding PancakeSwap V2 LP token using the addLiquidity function of the PancakeSwap router. Finally, it sends any remaining tokens back to the caller using the \_safeTransfer function of the IERC20 interface.
3. \_removeLiquidity(address \_lp,address \_tokenA,address \_tokenB, uint \_amount) internal: This internal function is used to remove liquidity from the Uniswap V2 LP token. The function takes in the following inputs:  
    \_lp: The address of the Uniswap V2 LP token.  
    \_tokenA: The address of the first token in the pair.  
    \_tokenB: The address of the second token in the pair.  
    \_amount: The amount of Uniswap V2 LP tokens to remove.  
    The function transfers the Uniswap V2 LP tokens from the caller to the contract using the safeTransferFrom function of the IERC20 interface. It then approves the PancakeSwap router to spend the specified amount of Uniswap V2 LP tokens and removes liquidity from the Uniswap V2 LP token using the removeLiquidity function of the PancakeSwap router.

#RewardsDistributor.sol

This is a Solidity smart contract for a rewards distributor, which facilitates the distribution of rewards for users who hold ve(3,3) tokens. The contract contains several functions to manage the distribution of rewards and track the balance of tokens.

Variables:

* start\_time: The start time of the rewards distribution, rounded down to the nearest week.
* time\_cursor: The current time cursor, used to track the ve(3,3) supply.
* time\_cursor\_of: A mapping of timestamps to time cursors.
* user\_epoch\_of: A mapping of user tokens to user epochs.
* last\_token\_time: The timestamp of the last token checkpoint.
* tokens\_per\_week: An array of token distributions per week.
* token\_last\_balance: The balance of the token at the last checkpoint.
* ve\_supply: An array of ve(3,3) supplies per week.
* owner: The owner of the contract.
* voting\_escrow: The address of the voting escrow contract.
* token: The address of the token contract.
* depositor: The address of the depositor, which is allowed to call the checkpoint\_token function

Functions:

1. timestamp() external view returns (uint): This function returns the current timestamp, rounded down to the nearest week.
2. \_checkpoint\_token() internal: This internal function checkpoints the token balance, and updates the tokens\_per\_week array with the new distribution of tokens since the last checkpoint. It emits a CheckpointToken event to log the checkpoint.
3. checkpoint\_token() external: This function calls \_checkpoint\_token() and is only callable by the depositor address.
4. \_find\_timestamp\_epoch(address ve, uint \_timestamp) internal view returns (uint): This internal function finds the epoch for a given timestamp by binary search using the point\_history function of the IVotingEscrow interface.
5. \_find\_timestamp\_user\_epoch(address ve, uint tokenId, uint \_timestamp, uint max\_user\_epoch) internal view returns (uint): This internal function finds the user epoch for a given timestamp and tokenId by binary search using the user\_point\_history function of the IVotingEscrow interface.
6. ve\_for\_at(uint \_tokenId, uint \_timestamp) external view returns (uint): This function finds the ve(3,3) balance of a user at a given timestamp by calling \_find\_timestamp\_user\_epoch and computing the balance using the bias and slope values from the user's point\_history.
7. \_checkpoint\_total\_supply() internal: This internal function checkpoints the total ve(3,3) supply, and updates the ve\_supply array with the new distribution of tokens since the last checkpoint. It calls the checkpoint function of the IVotingEscrow interface to update the ve(3,3) balances.
8. \_checkpoint\_total\_supply() internal: This internal function is used to checkpoint the total ve(3,3) supply and update the ve\_supply array with the new distribution of tokens since the last checkpoint. It first retrieves the address of the voting escrow contract, the current time cursor, and the rounded timestamp. It then calls the checkpoint() function of the IVotingEscrow interface to update the ve(3,3) balances. Next, it loops through the timestamps up to 20 times and updates the ve\_supply array with the new distribution of tokens since the last checkpoint.
9. checkpoint\_total\_supply() external: This external function calls the \_checkpoint\_total\_supply() internal function and is only callable by the contract owner.
10. claimable(uint \_tokenId) external view returns (uint): This external function returns the amount of rewards that a user is eligible to claim for a given token ID. It calls the \_claimable() internal function to calculate the rewards, passing in the voting\_escrow address and the last token time.
11. claim(uint \_tokenId) external returns (uint): This external function allows a user to claim their rewards for a given token ID. It first checks if the current block timestamp is greater than or equal to the time\_cursor and calls the \_checkpoint\_total\_supply() internal function to update the ve(3,3) supply if necessary. It then calls the \_claim() internal function to calculate the rewards, passing in the voting\_escrow address and the last token time. If the rewards are not zero, the function checks if the token is locked and sends the rewards directly to the token owner if the lock has ended, or deposits the rewards into the voting escrow contract if the lock is still active. Finally, the function updates the token\_last\_balance variable and returns the amount of rewards claimed.
12. claim\_many(uint[] memory \_tokenIds) external returns (bool): This external function allows a user to claim rewards for multiple token IDs at once. It first checks if the current block timestamp is greater than or equal to the time\_cursor and calls the \_checkpoint\_total\_supply() internal function to update the ve(3,3) supply if necessary. It then loops through each token ID in the input array and calls the \_claim() internal function to calculate the rewards, passing in the voting\_escrow address and the last token time. If the rewards are not zero, the function checks if the token is locked and sends the rewards directly to the token owner if the lock has ended, or deposits the rewards into the voting escrow contract if the lock is still active. Finally, the function updates the token\_last\_balance variable and returns true.
13. setDepositor(address \_depositor) external: This external function allows the contract owner to set the depositor address, which is used to deposit rewards into the voting escrow contract.
14. setOwner(address \_owner) external: This external function allows the contract owner to transfer ownership of the contract to a new address.
15. withdrawERC20(address \_token) external: This external function allows the contract owner to withdraw any ERC20 tokens that have been deposited into the contract. It requires that the caller is the contract owner, and it transfers the full balance of the specified ERC20 token to the caller's address.

#Router.sol

This contract is a Solidity implementation of a router for a decentralized exchange (DEX) on the Ethereum blockchain. It allows users to swap tokens and add or remove liquidity to/from liquidity pools.

Functions:

1. constructor(address \_factory, address \_weth): Initializes the contract with the addresses of the factory contract and the WETH (Wrapped Ether) contract.
2. receive() external payable: A fallback function that accepts Ether payments only from the WETH contract
3. sortTokens(address tokenA, address tokenB) public pure returns (address token0, address token1): Sorts two token addresses in ascending order.
4. pairFor(address tokenA, address tokenB, bool stable) public view returns (address pair): Calculates the address of the liquidity pool contract for a given token pair.
5. quoteLiquidity(uint amountA, uint reserveA, uint reserveB) internal pure returns (uint amountB): Calculates the equivalent amount of Token B for a given amount of Token A and the reserves of a liquidity pool.
6. getReserves(address tokenA, address tokenB, bool stable) public view returns (uint reserveA, uint reserveB): Fetches and returns the reserves of a liquidity pool for a given token pair.
7. getAmountOut(uint amountIn, address tokenIn, address tokenOut) external view returns (uint amount, bool stable): Calculates the amount of Token Out that a user will receive for a given amount of Token In.
8. getAmountsOut(uint amountIn, route[] memory routes) public view returns (uint[] memory amounts): Calculates the amounts of output tokens that a user will receive for a given input token, as well as the path to take through the liquidity pools.
9. isPair(address pair) external view returns (bool): Checks if a given address is a valid liquidity pool contract.
10. quoteAddLiquidity(address tokenA, address tokenB, bool stable, uint amountADesired, uint amountBDesired) external view returns (uint amountA, uint amountB, uint liquidity): Calculates the amount of liquidity tokens that will be minted for a given amount of Token A and Token B added to a liquidity pool.
11. addLiquidity(address tokenA, address tokenB, bool stable, uint amountADesired, uint amountBDesired, uint amountAMin, uint amountBMin, address to, uint deadline) external payable returns (uint amountA, uint amountB, uint liquidity): Adds liquidity to a given pool by transferring the desired amount of both tokens to the contract and minting liquidity tokens.
12. removeLiquidity(address tokenA, address tokenB, bool stable, uint liquidity, uint amountAMin, uint amountBMin, address to, uint deadline) external returns (uint amountA, uint amountB): Removes liquidity from a given pool by burning liquidity tokens and transferring the corresponding amounts of both tokens to the user.
13. swapExactTokensForTokens(uint amountIn, uint amountOutMin, route[] memory routes, address to, uint deadline) external: Swaps a given amount of input tokens for output tokens by finding the best path through the liquidity pools.
14. swapTokensForExactTokens(uint amountOut, uint amountInMax, route[] memory routes, address to, uint deadline) external: Swaps input tokens for a given amount of output tokens by finding the best path through the liquidity pools.
15. swapExactETHForTokens(uint amountOutMin, route[] memory routes, address to, uint deadline) external payable: Swaps Ether for output tokens by finding the best path through the liquidity pools.
16. swapTokensForExactETH(uint amountOut, uint amountInMax, route[] memory routes, address to, uint deadline) external: Swaps input tokens for a given amount of Ether by finding the best path through the liquidity pools.
17. swapExactTokensForETH(uint amountIn, uint amountOutMin, route[] memory routes, address to, uint deadline) external: Swaps a given amount of input tokens for Ether by finding the best path through the liquidity pools.
18. quoteRemoveLiquidity(address tokenA, address tokenB, bool stable, uint liquidity) external view returns (uint amountA, uint amountB):  
    This function takes in the addresses of two tokens, a boolean value indicating whether the tokens are stablecoins, and a liquidity amount, and returns the amounts of each token that would be received upon removing that amount of liquidity from the pool.
19. \_addLiquidity(address tokenA, address tokenB, bool stable, uint amountADesired, uint amountBDesired, uint amountAMin, uint amountBMin) internal returns (uint amountA, uint amountB):  
    This internal function takes in the addresses of two tokens, a boolean value indicating whether the tokens are stablecoins, and the desired and minimum amounts of each token to be added to the pool. It returns the actual amounts of the tokens that will be added.
20. addLiquidity(address tokenA, address tokenB, bool stable, uint amountADesired, uint amountBDesired, uint amountAMin, uint amountBMin, address to, uint deadline) external ensure(deadline) returns (uint amountA, uint amountB, uint liquidity):  
    This function allows users to add liquidity to the pool using the \_addLiquidity function. It takes in the same inputs as \_addLiquidity, as well as the recipient address and a deadline for the transaction. It returns the actual amounts of the tokens added and the resulting liquidity tokens.
21. addLiquidityETH(address token, bool stable, uint amountTokenDesired, uint amountTokenMin, uint amountETHMin, address to, uint deadline) external payable ensure(deadline) returns (uint amountToken, uint amountETH, uint liquidity):  
    This function allows users to add liquidity to the pool using Ether as one of the tokens. It takes in the address of the non-Ether token, a boolean value indicating whether the token is a stablecoin, and the desired and minimum amounts of the non-Ether token to be added to the pool. It also takes in the minimum amount of Ether to be added and a deadline for the transaction. It returns the actual amounts of the tokens added and the resulting liquidity tokens.
22. removeLiquidity(address tokenA, address tokenB, bool stable, uint liquidity, uint amountAMin, uint amountBMin, address to, uint deadline) public ensure(deadline) returns (uint amountA, uint amountB):  
    This function allows users to remove liquidity from the pool. It takes in the addresses of the two tokens, a boolean value indicating whether the tokens are stablecoins, the amount of liquidity to be removed, the minimum amounts of each token to be received, the recipient address, and a deadline for the transaction. It returns the actual amounts of each token received upon removing the specified liquidity.
23. removeLiquidityETH(address token, bool stable, uint liquidity, uint amountTokenMin, uint amountETHMin, address to, uint deadline) public ensure(deadline) returns (uint amountToken, uint amountETH):  
    This function allows users to remove liquidity from the pool where one of the tokens is Ether. It takes in the address of the non-Ether token, a boolean value indicating whether the token is a stablecoin, the amount of liquidity to be removed, the minimum amounts of the non-Ether token and Ether to be received, the recipient address, and a deadline for the transaction. It returns the actual amounts of the non-Ether token and Ether received upon removing the specified liquidity.
24. removeLiquidityWithPermit(address tokenA, address tokenB, bool stable, uint liquidity, uint amountAMin, uint amountBMin, address to, uint deadline, bool approveMax, uint8 v, bytes32 r, bytes32 s) external returns (uint amountA, uint amountB):  
    This function allows users to remove liquidity from the pool by calling the removeLiquidity function after approving the transaction using the permit function. It takes in the addresses of the two tokens, a boolean value indicating whether the tokens are stablecoins, the amount of liquidity to be removed, the minimum amounts of each token to be received, the recipient address, a deadline for the transaction, a boolean value indicating whether to approve the maximum amount of liquidity, and the v, r, and s values from the permit signature. It returns the actual amounts of each token received upon removing the specified liquidity.
25. removeLiquidityETHWithPermit(address token, bool stable, uint liquidity, uint amountTokenMin, uint amountETHMin, address to, uint deadline, bool approveMax, uint8 v, bytes32 r, bytes32 s) external returns (uint amountToken, uint amountETH):  
    This function allows users to remove liquidity from the pool where one of the tokens is Ether by calling the removeLiquidityETH function after approving the transaction using the permit function. It takes in the address of the non-Ether token, a boolean value indicating whether the token is a stablecoin, the amount of liquidity to be removed, the minimum amounts of the non-Ether token and Ether to be received, the recipient address, a deadline for the transaction, a boolean value indicating whether to approve the maximum amount of liquidity, and the v, r, and s values from the permit signature. It returns the actual amounts of the non-Ether token and Ether received upon removing the specified liquidity.
26. \_swap(uint[] memory amounts, route[] memory routes, address \_to) internal virtual:  
    This internal function performs a series of swaps between different pairs of tokens to convert one token to another. It takes in an array of amounts representing the amount of each token to be swapped, an array of route structs representing the pairs of tokens to be swapped and whether they are stablecoins, and the recipient address.
27. swapExactTokensForTokensSimple(uint amountIn, uint amountOutMin, address tokenFrom, address tokenTo, bool stable, address to, uint deadline) external ensure(deadline) returns (uint[] memory amounts):  
    This function allows users to swap a fixed amount of one token for another token. It takes in the amount of the input token to be swapped, the minimum amount of output token to be received, the address of the input token, the address of the output token, a boolean value indicating whether the tokens are stablecoins, the recipient address, and a deadline for the transaction. It returns an array of amounts representing the amount of each token received upon completing the swap.
28. swapExactTokensForTokens(uint amountIn, uint amountOutMin, route[] calldata routes, address to, uint deadline) external ensure(deadline) returns (uint[] memory amounts):  
    This function allows users to swap a fixed amount of one token for another token using a custom route of multiple pairs. It takes in the amount of the input token to be swapped, the minimum amount of output token to be received, an array of route structs representing the pairs of tokens to be swapped and whether they are stablecoins, the recipient address, and a deadline for the transaction. It returns an array of amounts representing the amount of each token received upon completing the swap.
29. swapExactETHForTokens(uint amountOutMin, route[] calldata routes, address to, uint deadline) external payable ensure(deadline) returns (uint[] memory amounts):  
    This function allows users to swap Ether for another token using a custom route of multiple pairs. It takes in the minimum amount of output token to be received, an array of route structs representing the pairs of tokens to be swapped and whether they are stablecoins, the recipient address, and a deadline for the transaction. It returns an array of amounts representing the amount of each token received upon completing the swap.
30. swapExactTokensForETH(uint amountIn, uint amountOutMin, route[] calldata routes, address to, uint deadline) external ensure(deadline) returns (uint[] memory amounts):  
    This function allows users to swap a token for Ether using a custom route of multiple pairs. It takes in the amount of the input token to be swapped, the minimum amount of Ether to be received,an array of route structs representing the pairs of tokens to be swapped and whether they are stablecoins, the recipient address, and a deadline for the transaction. It returns an array of amounts representing the amount of each token received upon completing the swap.
31. swapExactTokensForTokensSupportingFeeOnTransferTokens(uint amountIn, uint amountOutMin, route[] calldata routes, address to, uint deadline) external ensure(deadline) returns (uint[] memory amounts):  
    This function allows users to swap a fixed amount of one token for another token, even if one or both of the tokens have a transfer fee. It takes in the amount of the input token to be swapped, the minimum amount of output token to be received, an array of route structs representing the pairs of tokens to be swapped and whether they are stablecoins, the recipient address, and a deadline for the transaction. It returns an array of amounts representing the amount of each token received upon completing the swap.
32. addLiquidity(address tokenA, address tokenB, bool stable, uint amountADesired, uint amountBDesired, uint amountAMin, uint amountBMin, address to, uint deadline) external returns (uint amountA, uint amountB, uint liquidity):  
    This function allows users to add liquidity to the pool by providing an equal value of both tokens. It takes in the addresses of the two tokens, a boolean value indicating whether the tokens are stablecoins, the desired amount of each token to be added, the minimum amounts of each token to be added, the recipient address, and a deadline for the transaction. It returns the actual amounts of each token added and the amount of liquidity created.
33. addLiquidityETH(address token, bool stable, uint amountTokenDesired, uint amountTokenMin, uint amountETHMin, address to, uint deadline) external payable returns (uint amountToken, uint amountETH, uint liquidity):  
    This function allows users to add liquidity to the pool where one of the tokens is Ether by providing an equal value of Ether and the non-Ether token. It takes in the address of the non-Ether token, a boolean value indicating whether the token is a stablecoin, the desired amount of the non-Ether token to be added, the minimum amounts of the non-Ether token and Ether to be added, the recipient address, and a deadline for the transaction. It returns the actual amounts of the non-Ether token and Ether added and the amount of liquidity created.
34. permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, bytes32 s) external:  
    This function allows a token owner to give permission for a spender to spend their tokens on their behalf by signing a message. It takes in the token owner's address, the spender's address, the amount of tokens to be spent, a deadline for the permission to be used, and the v, r, and s values from the signature.

#Royalties.sol

This contract is a smart contract on the Ethereum blockchain that implements a royalties distribution mechanism for a specific token called "Echo". The contract allows users to deposit a certain amount of the WBNB token into the contract, which is then used to calculate the amount of Echo tokens that each user can claim as royalties.

Variables:

* wbnb: An instance of the ERC20 WBNB token contract.
* DISTRIBUTION: A constant that represents the duration of a distribution period (7 days).
* epoch: A variable that represents the current distribution epoch.
* echoholders: An instance of the EchoHolders contract, which provides information about the total supply of the Echo token and the amount of reserved tokens.
* owner: An address variable that represents the owner of the contract.
* feesPerEpoch: A mapping that stores the amount of WBNB tokens deposited in each epoch.
* totalSupply: A mapping that stores the total supply of Echo tokens in each epoch.
* reservedAmounts: A mapping that stores the amount of reserved Echo tokens in each epoch.
* depositors: A mapping that stores whether an address is allowed to deposit funds into the contract.
* userCheckpoint: A mapping that stores the epoch at which each user last claimed royalties.

Functions:

1. deposit(uint256 amount) external payable allowed: Allows users to deposit WBNB tokens into the contract. If the user sends WBNB tokens, they are transferred to the contract using the safeTransferFrom function. If the user sends ETH, they are converted to WBNB tokens using the IWETH contract's deposit function. The amount of WBNB tokens deposited is stored in the feesPerEpoch mapping, and the total supply and reserved amount of Echo tokens are stored in the respective mappings. The epoch variable is incremented to mark the start of a new distribution period.
2. withdrawERC20(address \_token) external onlyOwner: Allows the contract owner to withdraw any ERC20 tokens that have been deposited into the contract.
3. claim(address to) external nonReentrant: Allows users to claim their share of Echo tokens as royalties. The amount of Echo tokens that each user can claim is calculated using the claimable function, and these tokens are transferred to the user's address using the safeTransfer function. The userCheckpoint mapping is updated to mark the epoch at which the user claimed royalties.
4. claimable(address user) public view returns(uint): Calculates the amount of Echo tokens that a user can claim as royalties. The function calculates the total fees in WBNB tokens that were deposited in each epoch, and then calculates the user's share of the fees based on the number of Echo tokens that the user contributed to the total supply in each epoch. The function returns the total amount of Echo tokens that the user can claim.
5. setDepositor(address depositor) external onlyOwner: Allows the contract owner to add a new depositor that is allowed to deposit funds into the contract.
6. removeDepositor(address depositor) external onlyOwner: Allows the contract owner to remove a depositor that is no longer allowed to deposit funds into the contract.
7. setOwner(address \_owner) external onlyOwner: Allows the contract owner to change the ownership of the contract to a new address.
8. receive() external payable: A fallback function that allows the contract to receive ETH payments. If the function is called with ETH, it is converted to WBNB tokens using the IWETH contract's deposit function.

#SimpleAirdropDAO.sol

This is a Solidity smart contract named SimpleAirdropDAO that allows an owner (or a second owner) to distribute an airdrop to a list of specified users. Here is a breakdown of its functions:

1. setAmountPerUser(uint256 \_amount) - This function allows the owner to set the amount of tokens to be distributed to each user.
2. distributeAirdrop() - This function allows the owner to distribute the airdrop to the users in the list.
3. withdrawERC20(address \_token) - This function allows the owner to withdraw any ERC20 tokens that may have been sent to the contract by mistake.
4. setOwner(address \_owner) - This function allows the owner to change their own address.
5. withdrawERC20(address \_token) - This function allows the owner to withdraw any ERC20 tokens that may have been sent to the contract by mistake.
6. setOwner2(address \_owner) - This function allows the owner to set a second owner's address.

#StakingNFTFeeConverter.sol

This is a Solidity smart contract named StakingNFTFeeConverter that involves staking and distributing rewards. Here is a breakdown of its functions:

1. claimFees() - This function allows a keeper (the contract owner or a designated address) to claim staking fees from all pairs in the contract.
2. claimSingleFee(address \_pair) - This function allows a keeper to claim staking fees from a single specified pair in the contract.
3. swapManual(uint amountIn,uint amountOutMin, IRouter01.route[] calldata \_routes,uint deadline) - This function allows a keeper to manually swap any token in the contract using a specified route.
4. setDistribution() - This function allows a keeper to set the distribution rate for the rewards generated by staking WBNB.
5. swap() - This function allows a keeper to automatically swap any token in the contract using its stored routes.
6. setPair(address[] memory \_\_pairs) - This function allows the owner to set pairs for the contract.
7. setToken(address pair) - This internal function is called by other functions to add a token used in a pair to the contract's list of tokens, and to define a default route to convert the token to the Wrapped Binance Coin (WBNB) token.
8. removeToken(address token) - This function allows the contract owner to remove a token from the contract's list of tokens.
9. addToken(address token, IRouter01.route memory routes) - This function allows the contract owner to manually add a token to the contract's list of tokens, along with a specified route to convert the token to WBNB.
10. setRoutesFor(address token, IRouter01.route memory routes) - This function allows the contract owner to update the conversion routes for a specified token.
11. withdrawERC20(address \_token) - This function allows the contract owner to withdraw any ERC20 token that may be stuck in the contract.
12. \_tokens() - This function returns an array of all tokens currently used in the contract.
13. \_pairs() - This function returns an array of all pairs currently used in the contract.
14. transferOwnership(address newOwner) - This function allows the contract owner to transfer ownership of the contract to a new address.
15. setKeeper(address \_keeper) - This function allows the contract owner to add a new keeper to the contract. A keeper is an address that has permission to perform certain actions on the contract.
16. removeKeeper(address \_keeper) - This function allows the contract owner to remove a keeper from the contract.
17. setRouter(address \_router) - This function allows the contract owner to set the address of the router used in the DEX/AMM system.
18. setMasterchef(address \_masterchef) - This function allows the contract owner to set the address of the Masterchef contract used in the DEX/AMM system.
19. setPairFactory(address \_pairFactory) - This function allows the contract owner to set the address of the pair factory used in the DEX/AMM system.
20. \_safeTransfer(address token,address to,uint256 value) - This internal function performs a safe transfer of an ERC20 token to a specified address.
21. \_safeApprove(address token,address spender,uint256 value) - This internal function performs a safe approval of an ERC20 token to a specified spender.

#VeArtProxy.sol

This is a Solidity smart contract that implements the IVeArtProxy interface, and defines two functions: toString and \_tokenURI.

The toString function is an internal pure function that takes an unsigned integer and returns its string representation. It is used by the \_tokenURI function to convert various uint values to string format.

The \_tokenURI function takes four uint arguments: \_tokenId, \_balanceOf, \_locked\_end, and \_value. It returns a string that represents a token's metadata URI, which conforms to the OpenSea metadata standard. The function generates an SVG image that displays the four arguments in text format and encodes it in Base64. The Base64-encoded SVG image is then used to generate the JSON metadata for the token. The JSON metadata includes the name, description, and image of the token, and is also encoded in Base64. Finally, the function concatenates the JSON metadata with the appropriate prefix and returns the complete metadata URI as a string.

Neither of these functions modifies the state of the contract or interacts with external contracts, as they are both internal pure functions.

#Voter.sol

This is a Solidity smart contract named "Voter" that implements the IVoter interface. The contract is used for weighted voting on a set of whitelisted pools with incentives provided by internal and external bribes. The following are the functions and variables in this contract:

Variables:

* \_ve: the address of the ve token that governs these contracts.
* factory: the address of the PairFactory contract.
* base: the address of the token of the VotingEscrow contract.
* gaugefactory: the address of the IGaugeFactory contract.
* bribefactory: the address of the IBribeFactory contract.
* DURATION: a constant that represents the duration of an epoch.
* minter: the address of the contract that is allowed to whitelist tokens.
* governor: the address of the IGovernor contract.
* emergencyCouncil: the address of a credibly neutral party similar to Curve's Emergency DAO.
* totalWeight: the total voting weight of all the voters.
* pools: an array that contains all the pools that are viable for incentives.
* gauges: a mapping from the pool address to the gauge address.
* poolForGauge: a mapping from the gauge address to the pool address.
* internal\_bribes: a mapping from the gauge address to the address of the internal bribe contract.
* external\_bribes: a mapping from the gauge address to the address of the external bribe contract.
* weights: a mapping from the pool address to the voting weight of the pool.
* votes: a nested mapping from the NFT token ID to the pool address to the voting weight of the pool.
* poolVote: a nested mapping from the NFT token ID to an array of pool addresses that the voter has voted for.
* usedWeights: a mapping from the NFT token ID to the total voting weight of the user.
* lastVoted: a mapping from the NFT token ID to the timestamp of the last vote, to ensure one vote per epoch.
* isGauge: a mapping from the gauge address to a boolean value that indicates whether the gauge is valid.
* isWhitelisted: a mapping from the token address to a boolean value that indicates whether the token is whitelisted.
* isAlive: a mapping from the address to a boolean value that indicates whether the address is valid.

Functions:

1. constructor: initializes the contract with addresses of the ve token, PairFactory contract, IGaugeFactory contract, and IBribeFactory contract.
2. lock: a modifier that prevents re-entrancy attacks.
3. onlyNewEpoch: a modifier that ensures that the user has not already voted in the current epoch.
4. initialize: initializes the contract with an array of whitelisted tokens and the address of the minter.
5. setGovernor: sets the address of the IGovernor contract.
6. setEmergencyCouncil: sets the address of the emergency council.
7. reset: resets the voting weights of an NFT token and abstains from voting.
8. \_reset: internal function that resets the voting weights of an NFT token and removes the user's vote.
9. poke: updates the voting weights of an NFT token without changing the user's vote.
10. \_vote: internal function that sets the voting weights of an NFT token for a set of pools.
11. vote: sets the voting weights of an NFT token for a set of pools.
12. whitelist: adds a token to the whitelist.
13. createGauge(address \_pool) external returns (address): This function creates a new liquidity gauge for the specified pool. It takes the pool address as input and returns the address of the newly created gauge.
14. killGauge(address \_gauge) external: This function is used by the emergency council to kill a gauge, making it no longer active. It takes the address of the gauge to be killed as input.
15. reviveGauge(address \_gauge) external: This function is used by the emergency council to revive a previously killed gauge, making it active again. It takes the address of the gauge to be revived as input.
16. attachTokenToGauge(uint tokenId, address account) external: This function is used to attach a voting token to a gauge. It takes the ID of the voting token and the account address as input.
17. emitDeposit(uint tokenId, address account, uint amount) external: This function emits a deposit event for the specified gauge. It takes the ID of the voting token, the account address, and the deposit amount as input.
18. detachTokenFromGauge(uint tokenId, address account) external: This function is used to detach a voting token from a gauge. It takes the ID of the voting token and the account address as input.
19. emitWithdraw(uint tokenId, address account, uint amount) external: This function emits a withdraw event for the specified gauge. It takes the ID of the voting token, the account address, and the withdraw amount as input.
20. length() external view returns (uint): This function returns the number of pools that have gauges.
21. poolVoteLength(uint tokenId) external view returns(uint): This function returns the number of gauges that a voting token has been used to vote on.
22. notifyRewardAmount(uint amount) external: This function is used to distribute rewards to gauges. It takes the amount of rewards to distribute as input.
23. updateFor(address[] memory \_gauges) external: This function updates the accumulated rewards for the specified gauges. It takes an array of gauge addresses as input.
24. updateForRange(uint start, uint end) public: This function updates the accumulated rewards for gauges in a specified range. It takes the start and end indices of the range as input.
25. updateAll() external: This function updates the accumulated rewards for all gauges.
26. updateGauge(address \_gauge) external: This function updates the accumulated rewards for a single gauge.
27. claimRewards(address[] memory \_gauges, address[][] memory \_tokens) external: This function is used to claim rewards for the specified gauges and with specified tokens. It takes two arrays as input: an array of gauge addresses and an array of arrays of token addresses.
28. claimBribes(address[] memory \_bribes, address[][] memory \_tokens, uint \_tokenId) external: This function is used to claim bribes for the specified bribes and with specified tokens. It takes three arrays as input: an array of bribe addresses, an array of arrays of token addresses, and the ID of the voting token.
29. claimFees(address[] memory \_fees, address[][] memory \_tokens, uint \_tokenId) external: This function is used to claim fees for the specified fees and with specified tokens. It takes three arrays as input: an array of fee addresses, an array of arrays of token addresses, and the ID of the voting token.
30. distributeFees(address[] memory \_gauges) external: This function is used to distribute fees to the specified gauges. It takes an array of gauge addresses as input.
31. distribute(address \_gauge) public lock: This function is used to distribute rewards to a specified gauge. It takes the address of the gauge as input.
32. distributeAll() external: This function is used to distribute rewards to all gauges.
33. distribute(uint start, uint finish) public: This function is used to distribute rewards to gauges in a specified range. It takes the start and end indices of the range as input.
34. distribute(address[] memory \_gauges) external: This function is used to distribute rewards to the specified gauges. It takes an array of gauge addresses as input.

#VotingEscrow.sol

This contract is a voting escrow implementation that allows users to escrow ERC-20 tokens in the form of an ERC-721 NFT. The NFT represents the user's locked balance, and the weight of the user's vote depends on the amount of time the tokens are locked for.

Here is a brief description of the functions in the contract, their inputs, and what they return:

1. Deposit: This function is used to deposit ERC-20 tokens into the escrow by minting a new ERC-721 token. It takes five inputs: the provider's address, the value of tokens to be locked, the lock time, the deposit type, and the timestamp. It emits a Deposit event.
2. Withdraw: This function is used to withdraw tokens from the escrow by burning the ERC-721 token. It takes three inputs: the provider's address, the value of tokens to be withdrawn, and the timestamp. It emits a Withdraw event.
3. Supply: This function is used to update the total supply of locked tokens. It takes two inputs: the previous supply and the new supply. It emits a Supply event.
4. setTeam: This function is used to set the team address, which requires the caller to be the current team address.
5. setArtProxy: This function is used to set the art proxy address, which requires the caller to be the current team address.
6. tokenURI: This function returns the URI metadata for a given ERC-721 token. It takes one input: the token ID. It returns a string.
7. ownerOf: This function returns the address of the owner of a given ERC-721 token. It takes one input: the token ID. It returns an address.
8. balanceOf: This function returns the number of ERC-721 tokens owned by a given address. It takes one input: the owner's address. It returns a uint256 value.
9. getApproved: This function returns the address of the approved account for a given NFT. It takes one input: the NFT ID. It returns an address.
10. isApprovedForAll: This function checks whether an operator is approved to manage all the NFTs owned by a given address. It takes two inputs: the owner's address and the operator's address. It returns a boolean value.
11. approve: This function sets the approved account for a given NFT. It takes two inputs: the address to be approved and the NFT ID. It emits an Approval event.
12. setApprovalForAll: This function enables or disables approval for an operator to manage all the NFTs owned by a given address. It takes two inputs: the operator's address and a boolean value indicating whether the operator is approved. It emits an ApprovalForAll event.
13. \_clearApproval: This is an internal function used to clear the approval of a given address for a given NFT. It takes two inputs: the owner's address and the NFT ID.
14. \_isApprovedOrOwner: This is an internal function used to determine whether a given address is approved to transfer a given NFT. It takes two inputs: the spender's address and the NFT ID. It returns a boolean value.
15. isApprovedOrOwner: This function is a public wrapper around the \_isApprovedOrOwner internal function. It takes two inputs: the spender's address and the NFT ID. It returns a boolean value.
16. \_transferFrom: This is an internal function used to transfer an NFT from one address to another. It takes four inputs: the current owner of the NFT (\_from), the new owner of the NFT (\_to), the ID of the NFT (\_tokenId), and the address of the sender (\_sender). The function first checks that the NFT is not attached or voted, then checks that the sender is approved to transfer the NFT, clears the approval, removes the NFT from the current owner, moves any delegate votes associated with the NFT, adds the NFT to the new owner, sets the block number for the ownership transfer, and emits a Transfer event.
17. transferFrom: This function is a public wrapper around the \_transferFrom internal function. It takes three inputs: the current owner of the NFT (\_from), the new owner of the NFT (\_to), and the ID of the NFT (\_tokenId). It calls the \_transferFrom function with the sender as msg.sender.
18. safeTransferFrom: This function is similar to transferFrom, but it also checks if the receiver is a smart contract and calls the onERC721Received function on the receiver if it exists. It takes four inputs: the current owner of the NFT (\_from), the new owner of the NFT (\_to), the ID of the NFT (\_tokenId), and additional data to send to the receiver (\_data). If the receiver is a smart contract, the function checks that the onERC721Received function exists and returns the correct value before transferring the NFT.
19. \_isContract: This is an internal function used to check if an address is a smart contract. It takes one input: the address to check. It returns a boolean value indicating whether the address is a smart contract.
20. supportsInterface: This function checks if the contract supports a given interface. It takes one input: the interface ID. It returns a boolean value indicating whether the contract supports the interface.
21. \_addTokenToOwnerList: This is an internal function used to add an NFT to a mapping of NFTs owned by a given address. It takes two inputs: the address of the receiver (\_to) and the ID of the NFT to be added (\_tokenId).
22. \_addTokenTo: This is an internal function used to add an NFT to a given address. It takes two inputs: the address of the receiver (\_to) and the ID of the NFT to be added (\_tokenId). It checks that the NFT is not already owned by someone, changes the owner of the NFT, updates the owner's token index tracking, and updates the owner's NFT count.
23. \_mint: This is an internal function used to mint a new NFT. It takes two inputs: the address of the receiver (\_to) and the ID of the new NFT to be minted (\_tokenId). It checks that the receiver address is not zero and adds the NFT to the receiver's ownership. It also emits a Transfer event indicating the NFT has been minted and transferred from address 0 to the receiver.
24. \_removeTokenFromOwnerList: This internal function removes an NFT from a list of NFTs owned by a given address. It takes two inputs: the address of the current owner of the NFT (\_from) and the ID of the NFT to be removed (\_tokenId). It updates two mappings: ownerToNFTokenIdList and tokenToOwnerIndex.
25. \_removeTokenFrom: This internal function removes an NFT from a given address. It takes two inputs: the address of the current owner of the NFT (\_from) and the ID of the NFT to be removed (\_tokenId). It checks that the sender is the current owner of the NFT, changes the owner of the NFT to address 0, updates the owner's token index tracking, and updates the owner's NFT count.
26. \_burn: This internal function burns (destroys) an NFT. It takes one input: the ID of the NFT to be burned (\_tokenId). It checks that the sender is the current owner or an approved operator of the NFT, clears the approval, removes the NFT from the current owner, moves any delegate votes associated with the NFT, emits a Transfer event indicating the NFT has been transferred from the current owner to address 0, and updates the total supply of NFTs.
27. get\_last\_user\_slope: This function returns the most recently recorded rate of voting power decrease for a given NFT. It takes one input: the ID of the NFT (\_tokenId). It returns a signed integer value representing the slope.
28. user\_point\_history\_\_ts: This function returns the timestamp for a specific checkpoint of a given NFT. It takes two inputs: the ID of the NFT (\_tokenId) and the index of the checkpoint (\_idx). It returns a uint value representing the epoch time of the checkpoint.
29. locked\_\_end: This function returns the epoch time when the lock on a given NFT ends. It takes one input: the ID of the NFT (\_tokenId). It returns a uint value representing the epoch time.
30. \_checkpoint: This internal function is used to calculate the voting power of a given NFT based on its locked balance and the slope of the voting power decay function. It takes four inputs: the ID of the NFT (\_tokenId), the previous locked balance of the NFT (old\_locked), the new locked balance of the NFT (new\_locked), and the current epoch value (epoch). It calculates the slope and bias of the decay function, updates the point history and slope changes for the NFT, and records the changed point and user history into the point history mapping.
31. \_deposit\_for: This internal function is used to deposit and lock tokens for a user. It takes five inputs: the ID of the NFT (\_tokenId), the amount of tokens to deposit (\_value), the new unlock time for the tokens (unlock\_time), the previous locked balance of the NFT (locked\_balance), and the type of deposit (deposit\_type). It updates the total supply of tokens, updates the locked balance of the NFT, and calls the \_checkpoint function to recalculate the voting power of the NFT.
32. checkpoint: This external function is used to record global data to a checkpoint. It calls the \_checkpoint function with the ID of 0, indicating that no specific NFT is being updated.
33. deposit\_for(uint \_tokenId, uint \_value) external nonreentrant: Allows a user to deposit a specified amount of tokens for a specific NFT and lock them. This function checks that the locked balance for the specified NFT is not zero and that the lock has not expired. It then calls the internal function \_deposit\_for with the appropriate inputs.
34. \_create\_lock(uint \_value, uint \_lock\_duration, address \_to) internal returns (uint): This internal function creates a new lock for a user by minting a new NFT, locking the specified amount of tokens, and calling the \_deposit\_for function with the appropriate inputs. The function then returns the ID of the newly minted NFT.
35. create\_lock(uint \_value, uint \_lock\_duration) external nonreentrant returns (uint): Allows a user to create a new lock for themselves by calling the internal function \_create\_lock with the appropriate inputs.
36. create\_lock\_for(uint \_value, uint \_lock\_duration, address \_to) external nonreentrant returns (uint): Allows a user to create a new lock for another user by calling the internal function \_create\_lock with the appropriate inputs.
37. increase\_amount(uint \_tokenId, uint \_value) external nonreentrant: Allows a user to increase the amount of tokens locked for a specific NFT. This function checks that the existing lock has not expired and then calls the internal function \_deposit\_for with the appropriate inputs.
38. increase\_unlock\_time(uint \_tokenId, uint \_lock\_duration) external nonreentrant: Allows a user to extend the unlock time for a specific NFT. This function checks that the existing lock has not expired and that the new unlock time is greater than the existing unlock time. It then calls the internal function \_deposit\_for with the appropriate inputs.
39. withdraw(uint \_tokenId) external nonreentrant: Allows a user to withdraw all locked tokens for a specific NFT. This function checks that the lock has expired and that there are no attachments or votes associated with the NFT. It then transfers the locked tokens to the user, burns the NFT, and emits an event indicating the withdrawal.
40. \_find\_block\_epoch(uint \_block, uint max\_epoch) internal view returns (uint): This internal function is used to estimate the timestamp for a given block number using binary search.
41. \_balanceOfNFT(uint \_tokenId, uint \_t) internal view returns (uint): This internal function is used to calculate the current voting power for a given NFT at a specified time. It retrieves the last point from the user's point history and subtracts the slope multiplied by the time elapsed since that point in time.
42. balanceOfNFT(uint \_tokenId) external view returns (uint): This function returns the current voting power for a given NFT at the current block timestamp.
43. balanceOfNFTAt(uint \_tokenId, uint \_t) external view returns (uint): This function returns the voting power for a given NFT at a specified timestamp.
44. \_balanceOfAtNFT(uint \_tokenId, uint \_block) internal view returns (uint): This internal function calculates the voting power of an NFT at a specific block number using binary search to retrieve the appropriate point history. It then calculates the voting power at the specified block using the slope and bias values from the retrieved point history.
45. balanceOfAtNFT(uint \_tokenId, uint \_block) external view returns (uint): This function returns the voting power of an NFT at a specific block number by calling the internal function \_balanceOfAtNFT with the appropriate inputs.
46. totalSupplyAt(uint \_block) external view returns (uint): This function calculates the total voting power at a specific block number by retrieving the appropriate point history and using the \_supply\_at internal function to calculate the total voting power at the specified block.
47. \_supply\_at(Point memory point, uint t) internal view returns (uint): This internal function calculates the total voting power at a specific time by iterating over each week from the starting point and calculating the slope and bias values using the slope changes and point history. The function then returns the total voting power at the specified time.
48. totalSupply() external view returns (uint): This function returns the current total voting power by calling totalSupplyAtT with the current block timestamp.
49. totalSupplyAtT(uint t) public view returns (uint): This function calculates the total voting power at a specific time by retrieving the appropriate point history and using the \_supply\_at internal function to calculate the total voting power at the specified time.
50. setVoter(address \_voter) external: This function sets the address of the voter who can vote on gauge proposals. Only the team can call this function.
51. voting(uint \_tokenId) external: This function allows the voter to vote on a specific NFT by setting the corresponding voted flag to true.
52. abstain(uint \_tokenId) external: This function allows the voter to abstain from voting on a specific NFT by setting the corresponding voted flag to false.
53. attach(uint \_tokenId) external: This function allows the voter to attach an NFT to a gauge proposal by incrementing the corresponding attachment counter.
54. detach(uint \_tokenId) external: This function allows the voter to detach an NFT from a gauge proposal by decrementing the corresponding attachment counter.
55. merge(uint \_from, uint \_to) external: This function allows the owner of two NFTs to merge them by transferring the locked tokens from one NFT to the other and burning the first NFT. This function also updates the point history to reflect the transfer.
56. split(uint[] memory amounts, uint \_tokenId) external: This function allows the owner of an NFT to split it into multiple NFTs and distribute the locked tokens among them. The amounts parameter is an array of weights for each new NFT, and the \_tokenId parameter is the ID of the original NFT. This function also updates the point history to reflect the split.
57. DOMAIN\_TYPEHASH and DELEGATION\_TYPEHASH: These are constants used to define the EIP-712 domain and delegation structs used by the contract.
58. \_delegates: This is a mapping of delegate addresses to delegator addresses.
59. MAX\_DELEGATES: This is a constant that limits the number of possible delegates to 1024.
60. checkpoints: This is a mapping of account addresses to checkpoints of delegated token balances. Each checkpoint includes an array of token IDs and corresponding balances.
61. numCheckpoints: This is a mapping of account addresses to the number of checkpoints they have.
62. nonces: This is a mapping of account addresses to nonces used for signing and validating signatures.
63. delegates(address delegator) public view returns (address): This function returns the delegate address for a given delegator. If the given delegator has not delegated, their own address is returned instead.
64. getVotes(address account) external view returns (uint): This function returns the total voting power of an account by summing up the balances of all NFTs owned by the account at the current block timestamp. It does this by retrieving the most recent checkpoint for the account and iterating over the token IDs in the checkpoint to calculate the total voting power.
65. getPastVotesIndex(address account, uint timestamp) public view returns (uint32): This function returns the index of the latest checkpoint for an account at a given timestamp. It does this by performing a binary search on the checkpoints array to find the checkpoint with the highest timestamp that is less than or equal to the given timestamp.
66. getPastVotes(address account, uint timestamp) public view returns (uint): This function returns the total voting power of an account at a given timestamp by summing up the balances of all NFTs owned by the account at that timestamp. It does this by calling getPastVotesIndex to find the index of the latest checkpoint at or before the given timestamp, and then iterating over the token IDs in that checkpoint to calculate the total voting power.
67. getPastTotalSupply(uint256 timestamp) external view returns (uint): This function returns the total supply of all NFTs at a given timestamp by calling the function totalSupplyAtT which is defined elsewhere in the contract.
68. \_moveTokenDelegates(address srcRep, address dstRep, uint \_tokenId) internal: This function updates the delegate mappings for a given NFT token when it is transferred from one owner to another. It does this by removing the token ID from the old delegate's list of tokens and adding it to the new delegate's list of tokens.
69. \_findWhatCheckpointToWrite(address account) internal view returns (uint32): This function returns the index of the checkpoint that should be written to when updating the token balances of an account. It does this by checking if the most recent checkpoint has the same timestamp as the current block timestamp, in which case it returns the index of the most recent checkpoint. Otherwise, it returns the index of the next available checkpoint.
70. \_moveAllDelegates(address owner, address srcRep, address dstRep) internal: This function updates the delegate mappings for all NFTs owned by a given owner when the owner's delegate changes. It does this by removing the NFTs owned by the owner from the old delegate's list of tokens and adding them to the new delegate's list of tokens.
71. \_delegate(address delegator, address delegatee) internal: This function delegates the voting power of a given delegator to a given delegatee. It does this by updating the delegate mapping for the delegator and emitting an event to signal the change. It also calls \_moveAllDelegates to update the delegate mappings for all NFTs owned by the delegator.
72. delegate(address delegatee) public: This function delegates the voting power of the caller to a given delegatee. If no delegatee is specified, it defaults to the caller's own address. It does this by calling \_delegate.
73. delegateBySig allows a user to delegate their voting power to another address by providing a signature.

#WrappedExternalBribe.sol

This is a Solidity smart contract that implements a bribe mechanism for incentivizing users to vote in a given pool. The contract is called WrappedExternalBribe and includes the following functions:

1. constructor: Initializes the contract with the address of the Voter contract, the address of an external bribe contract, and an array of whitelisted reward tokens.
2. lock: A modifier that prevents re-entrancy attacks by locking the function.
3. earned: Computes the amount of rewards earned by a given user for a given token and voting period.
4. getReward: Allows a user to claim rewards for a given token.
5. getRewardForOwner: Allows the Voter contract to claim rewards on behalf of a user.
6. lastTimeRewardApplicable: Returns the last time a reward was modified or the period ended.
7. left: Returns the amount of rewards remaining for a given token and voting period.
8. notifyRewardAmount: Notifies the contract of the amount of rewards available for a given token and voting period.
9. rewardsListLength: Returns the number of reward tokens.
10. swapOutRewardToken: Allows the team to swap out a reward token for another token.
11. \_bribeStart: Computes the start time of a voting period.
12. \_safeTransfer: Safely transfers tokens to a recipient.
13. \_safeTransferFrom: Safely transfers tokens from a sender to a recipient.