CD SECURITY

AUDIT REPORT

EBASE
August 2023

Introduction

A time-boxed security review of the **EBASE** protocol was done by **ddimitrov22** and **chrisdior4**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **EBASE**

EBASE is a simple ERC-20 token deployed on the BASE blockchain. Every individual who bridged ETH to BASE from the period of when the official bridge opened, to the night before BASE mainnet officially launched, was qualified for the airdrop.

Scope

The following smart contracts were in scope of the audit:

- EBASE.sol

The following number of issues were found, categorized by their severity:

- Critical & High: 4 issues
- Medium: 2 issues
- Low: 3 issues
- Informational: 7 issues

Severity classification

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

Impact - the technical, economic and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Findings Summary

ID	Title	Severity
[C-01]	Wrong router is hardcoded and `_transfer` will be bricked	Critical
[H-01]	Lack of slippage control can lead to sandwich attacks	High
[H-02]	Ineffective deadline check	High
[H-03]	Wrong total supply amount leads to problems with the fees	High
[M-01]	Use `call()` instead of `transfer()` when sending ETH	Medium
[M-02]	Not capped array can grow too big and lead to out of gas error	Medium
[L-01]	Avoid using the same name for variables	Low
[L-02]	`marketingWallet` and `devTaxWallet` are hardcoded to EOA addresses	Low
[L-03]	Use two-step ownership transfer approach	Low
[I-01]	Redundant code	Informational
[I-02]	Use the latest version of dependencies	Informational
[I-03]	Repetitive `require` statements can be made into modifier	Informational
[I-04]	Wrong revert message in `includeInReward()`	Informational
[I-05]	Missing event emissions in state changing methods	Informational
[I-06]	Prefer Solidity Custom Errors over `require` statements	Informational
[I-07]	Use newer Solidity version with a stable pragma statement	Informational

Detailed Findings

[C-01] Wrong router is hardcoded and `_transfer` will be bricked

Severity

Impact: High because a core function will be bricked

Likelihood: High because the router cannot be change because of the hardcoded address

Description

The `IUniswapV2Router02` interface is implemented and two of it's functions are called during a transfer - `swapExactTokensForETHSupportingFeeOnTransferTokens` and `addLiquidityETH`. The problem is that the hardcoded address for the router is wrong:

```
    IUniswapV2Router02 _uniswapV2Router =
IUniswapV2Router02(0xfCD3842f85ed87ba2889b4D35893403796e67FF1); //@audit -
this is the address of LeetSwapRouter on BASE
```

As the token is planned to be deployed on the BASE chain, the address for LeetSwapRouter is passed. This is problematic because the `addLiquidityETH` of LeetSwapRouter is different from UniswapV2Router:

```
    function addLiquidityETH(
        address token,
        uint256 amountTokenDesired,
        uint256 amountTokenMin,
        uint256 amountCANTOMin,
        address to,
        uint256 deadline
    )
        external
        payable
        ensure(deadline)
        returns (
            uint256 amountToken,
            uint256 amountCANTO,
            uint256 liquidity
        )
    {
        _startLiquidityManagement(token, address(wcanto));

        address pair = pairFor(token, address(wcanto));
        bool isStable = stablePairs[pair];
        (amountToken, amountCANTO) = _addLiquidity(
            token,
            address(wcanto),
            isStable,
            amountTokenDesired,
            msg.value,
            amountTokenMin,
            amountCANTOMin
        );
        _safeTransferFrom(token, msg.sender, pair, amountToken);
        wcanto.deposit{value: amountCANTO}();
        assert(wcanto.transfer(pair, amountCANTO));
        liquidity = ILeetSwapV2Pair(pair).mint(to);
        // refund dust eth, if any
        if (msg.value > amountCANTO) {
            _safeTransferETH(msg.sender, msg.value - amountCANTO);
        }
        _stopLiquidityManagement(token, address(wcanto));
    }

    function _safeTransferETH(address to, uint256 value) internal {
        (bool success, ) = to.call{value: value}(new bytes(0));
        if (!success) revert CantoTransferFailed();
    }
```

As you can see from the code snippet above the function works with wrapped Canto while the wrapped ETH is expected for the UniswapV2Router. This will lead to bricking the transfer function unless wCanto is provided every it is called.

Recommendations

There are 2 possible solutions here:

1. Use the actual UniswapV2Router.
2. Use the LeetSwapRouter interface but refactoring of the code will be required.

[H-01] Lack of slippage control can lead to sandwich attacks

Severity

Impact: High, as this will lead to loss of funds for users

Likelihood: Medium, since MEV is very prominent, the chance of that happening is pretty high

Description

The `amountOutMin` parameter in `swapExactTokensForETHSupportingFeeOnTransferTokens` is hard coded to 0 in `swapTokensForEth()`:

```
function swapTokensForEth(uint256 tokenAmount) private lockTheSwap {
    address[] memory path = new address[](2);
    path[0] = address(this);
    path[1] = uniswapV2Router.WETH();
    _approve(address(this), address(uniswapV2Router), tokenAmount);
    uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(
      tokenAmount,
      0,
      path,
      address(this),
      block.timestamp
    );
  }
```

This basically allows for 100% slippage as the call agrees to receive 0 amount of ETH for the swap. This can be done through a sandwich attack. The same applies to the `addLiquidity` function:

```
function addLiquidity(uint256 tokenAmount, uint256 ethAmount) private {
    _approve(address(this), address(uniswapV2Router), tokenAmount);

    uniswapV2Router.addLiquidityETH{value: ethAmount}(address(this),
tokenAmount, 0, 0, owner(), block.timestamp);
  }
```

This is a very easy target for MEV and bots to do a flash loan sandwich attack and can be done on every call if the trade transaction goes through a public mempool.

Recommendations

The best solution to this problem is to add an input parameter instead of hardcoding 0. The amountOutMin can be calculated off-chain and agreed upon by the user and can be passed to the call. This will protect the calls from sandwich attacks.

[H-02] Ineffective deadline check

Impact: High, because the transaction might be left hanging in the mempool and be executed way later than the user wanted at a possibly worse price

Likelihood: Medium, because there is a great chance that the user won't adjust the gas price to be lucrative for the validators to include its transaction fast

The deadline parameter in `swapExactTokensForETHSupportingFeeOnTransferTokens()` and `addLiquidityETH()` which are called in `swapTokensForEth()` and `addLiquidity()` is hard coded to `block.timestamp`.

Example in `addLiquidity()`:

```
function addLiquidity(uint256 tokenAmount, uint256 ethAmount) private {
    _approve(address(this), address(uniswapV2Router), tokenAmount);

    uniswapV2Router.addLiquidityETH{value: ethAmount}(address(this),
tokenAmount, 0, 0, owner(), block.timestamp); ///@audit here is the
problem
  }
```

The `addLiquidityETH()` in `UniswapV2Router02` contract:

```
function addLiquidityETH(
        address token,
        uint amountTokenDesired,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) external virtual override payable ensure(deadline) returns (uint
amountToken, uint amountETH, uint liquidity)
{
```

The `deadline` parameter enforces a time limit by which the transaction must be executed otherwise it will revert.

Lets take a look at a modifier which is present in the functions you are calling in `UniswapV2Router02` contract:

```
modifier ensure(uint deadline) {
        require(deadline >= block.timestamp, 'UniswapV2Router: EXPIRED');
        _;
    }
```

Now when the `deadline` is hardcoded as `block.timestamp`, the transaction will not revert because the require statement will always be fulfilled by block.timestamp == block.timestamp.

If a user chose a transaction fee that is too low for miners to be interested in including the transaction in a block, the transaction stays pending in the mempool for extended periods, which could be hours, days, weeks, or even longer.

This could lead to users getting a worse price, because a validator can just hold onto the transaction.

Recommendations

Protocols should let users who interact with AMMs set expiration deadlines. Without this, there's a risk of a serious loss of funds for anyone starting a swap, especially if there's no slippage parameter.

Use a user supplied deadline instead of `block.timestamp`.

[H-03] Wrong total supply amount leads to problems with the fees

Severity

Impact: Medium, as the fees will be miscalculated

Likelihood: High, as there are no mint or burn functions to regulate the total supply

Description

From the documentation and the tokenomics we can see that the total supply should be 10 billion tokens. However, the amount in the code is 100 million which is much less.

```
    uint256 private _tTotal = 100_000_000 * 10 ** 9;
```

This leads to problems with the fess, the max amount transactions and all of the limitations of the project. For example, the `_maxTxAmount` is required to be more than 10 million:

```
  function setMaxTxAmount(uint256 maxTxAmount) external onlyOwner {
    require(maxTxAmount > 10_000_000, "Max Tx Amount cannot be less than
10m");
    _maxTxAmount = maxTxAmount * 10 ** 9;
  }
```

If the total supply is 100 million then that is 10% of the total supply at minimum. If a transaction with such amount actually happens this will have a huge impact on the price of the token and will lead to massive slippage.

Recommendations

Change the total supply to the correct amount.

```diff
- uint256 private _tTotal = 100_000_000 * 10 ** 9;
+ uint256 private _tTotal = 10_000_000_000 * 10 ** 9;
```

[M-01] Use `call()` instead of `transfer()` when sending ETH

Severity

Impact: Medium, because if the recipient is a smart contract or multisig the trx will fail

Likelihood: Medium, because there is a big chance that the recipient will be a smart contract or a specific multisig wallet that requires more than 2300 gas

Description

Couple of functions in the contract are using the `transfer` method to withdraw accumulated or wrongly sent ETH in the contract to `marketingWallet`, `devTaxWallet`. These addresses are possible to be a smart contract that have a `receive` or `fallback` function that takes up more than the 2300 gas which is the limit of `transfer()`. Examples are some smart contract wallets or multi-sig wallets, so usage of `transfer` is discouraged.

More

Recommendations

Use a call with value instead of transfer.

[M-02] Not capped array can grow too big and lead to out of gas error

Severity

Impact: High because the contract will be in state of DoS

Likelihood: Low because it required too many accounts to be excluded

Description

The _getCurrentSupply and includeInReward functions both loop over the _excluded array to find out if there are any excluded accounts. The problem is that the array is not capped and can push unlimited number of accounts to the array. If an account needs to be removed, a loop over the whole array is needed inside includeInReward to pop the account:

```
 function includeInReward(address account) external onlyOwner {
  require(_isExcluded[account], "Account is already excluded");
  for (uint256 i = 0; i < _excluded.length; i++) { //@audit - this can
run out of gas if _excluded array is too big
   if (_excluded[i] == account) {
    _excluded[i] = _excluded[_excluded.length - 1];
    _tOwned[account] = 0;
    _isExcluded[account] = false;
    _excluded.pop();
    break;
   }
  }
```

If at some point there are a lot of excluded accounts in the array, iterating over them will be very costly and can result in a gas cost that is over the block gas limit. This will leave the contract in a state of DoS because the _getCurrentSupply is called in most of the core functions.

Recommendations

Limit the number of accounts tha can be excluded. Also, consider to cache the array length outside of the for loop to make the call more gas efficient.

[L-01] Avoid using the same name for variables

There are functions which take input parameters with the same name as storage variables which can lead to collision:

```
  function _approve(address owner, address spender, uint256 amount)
private {
    require(owner != address(0), "ERC20: approve from the zero address");
    require(spender != address(0), "ERC20: approve to the zero address");

    _allowances[owner][spender] = amount;
    emit Approval(owner, spender, amount);
  }
```

The `owner` variable is shadowed in this case as well in the `allowance` method. Consider changing the input parameter name to `_owner`.

[L-02] `marketingWallet` and `devTaxWallet` are hardcoded to EOA addresses

The two wallets that have the privilege to collect taxes and stuck balance are EOA addresses. This brings a centralisation risk because they can spend the money outside of the interest of the users. Moreover, if any or both of those accounts are compromised there is no way to return those roles. Consider adding methods with `onlyOwner` modifier that can change the accounts or use multi-sig wallets.

[L-03] Use two-step ownership transfer approach

The `owner` role is crucial for the protocol as there are a lot of functions with the `onlyOwner` modifier. Make sure to use a two-step ownership transfer approach by using `Ownable2Step` from OpenZeppelin as opposed to `Ownable` as it gives you the security of not unintentionally sending the `owner` role to an address you do not control. Also, consider using only `onlyOwner` modifier instead of using both `onlyOwner` and `restricted` modifiers because they are basically the same and using both only creates confusion.

[I-01] Redundant code

There is a lot of code that is not used anywhere which should be removed to make the code cleaner and more optimized:

```
interface IUniswapV2Pair {
    function DOMAIN_SEPARATOR() external view returns (bytes32);//@audit
not used
    function PERMIT_TYPEHASH() external pure returns (bytes32);//@audit
not used
    function nonces(address owner) external view returns (uint);//@audit
not used
```

```
    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, bytes32 s) external;//@audit not used
    event Mint(address indexed sender, uint amount0, uint amount1);//@audit not used
    event Burn(address indexed sender, uint amount0, uint amount1, address indexed to);//@audit not used
    event Sync(uint112 reserve0, uint112 reserve1);//@audit not used
    function MINIMUM_LIQUIDITY() external pure returns (uint);//@audit not used
    function getReserves() external view returns (uint112 reserve0, uint112 reserve1, uint32 blockTimestampLast);//@audit not used
    function price0CumulativeLast() external view returns (uint);//@audit not used
    function price1CumulativeLast() external view returns (uint);//@audit not used
    function kLast() external view returns (uint);//@audit not used
    function mint(address to) external returns (uint liquidity);//@audit not used
    function burn(address to) external returns (uint amount0, uint amount1);//@audit not used
    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external;//@audit not used
    function skim(address to) external;//@audit not used
    function sync() external;//@audit not used
    function initialize(address, address) external;//@audit not used
}
```

The 2 used functions from `IUniswapV2Factory` are `createPair` and `setFeeTo`. The rest can be removed.

The below functions from `IUniswapV2Router01` are not used and can be removed:

```

    function removeLiquidity
    function removeLiquidityETH
    function removeLiquidityWithPermit
    removeLiquidityETHWithPermit
    function swapExactTokensForTokens
    function swapTokensForExactTokens
    function swapExactETHForTokens
    function swapTokensForExactETH
    function swapExactTokensForETH
    function swapETHForExactTokens
    function quote
    function getAmountOut
    function getAmountIn
    function getAmountsOut
    function getAmountsIn
```

From `IUniswapV2Router02` interface the only used function is `swapExactTokensForETHSupportingFeeOnTransferTokens` and the rest can be removed. The _msgData function is not used and can be removed. There is no need to use SafeMath when compiler is ^0.8.0 because it has built-in under/overflow checks. Also you are both using methods from SafeMath and the normal arithmetic operators such as *, /, etc. Use them instead of SafeMath functions. The Address library is not used anywhere and can be removed.

[I-02] Use the latest version of dependencies

OpenZeppelin v4.7.0 Ownable library is used in the project while the latest available version is 4.9.0. Use the latest version to ensure the library is bug-free and optimized.

[I-03] Repetitive require statements can be made into modifier

The following require statement is used on 3 occasions within the contract:

```
require(_msgSender() == marketingWallet || _msgSender() == owner());
```

Turn it into modifier to make the code cleaner and more optimized

[I-04] Wrong revert message in includeInReward()

The error message is incorrect. Must be: Account is already included.

[I-05] Missing event emissions in state changing methods

It's a best practice to emit events on every state changing method for off-chain monitoring. The following methods are missing event emissions, which should be added:

- excludeFromFee()
- includeInFee()
- setMarketingWallet()
- setMaxTxAmount()
- setMaxWalletSize()
- setSwapThresholdAmount()
- addBotWallet()

- `removeBotWallet()`
- `allowtrading()`

[I-06] Prefer Solidity Custom Errors over `require` statements

Using Solidity Custom Errors has the benefits of less gas spent in reverted transactions, better interoperability of the protocol as clients of it can catch the errors easily on-chain, as well as you can give descriptive names of the errors without having a bigger bytecode or transaction gas spending, which will result in a better UX as well. Consider replacing the `require` statements with custom errors.

[I-07] Use newer Solidity version with a stable pragma statement

Using a floating pragma `^0.8.9` statement is discouraged as code can compile to different bytecodes with different compiler versions. Use a stable pragma statement to get a deterministic bytecode. Consider using a stable 0.8.19 version to make sure it is up to date.