EV Terminal Security Review

Pashov Audit Group

Conducted by: T1MOH, 0xdeadbeaf, 0xbepresent

April 17th 2024 - April 21st 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. 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.

3. Introduction

A time-boxed security review of the **Sage-Contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About EV Terminal

EV Terminal is the DEX platform implementing ERC-314 tokens. ERC314 is a derivative of ERC20 which aims to integrate a liquidity pool on the token in order to enable native swaps and reduce gas consumption.

Through the EV platform, users can swap between tokens, add liquidity, earn trading fees.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 136a71d9a862f556966544d9e167744740e6afd0

fixes review commit hash - 86e1bfa16ff1e4844850645c504ad1dc70f09061

Scope

The following smart contracts were in scope of the audit:

- `EvFactoryProxyOptFee`

7. Executive Summary

Over the course of the security review, T1MOH, 0xdeadbeaf, 0xbepresent engaged with EV Terminal to review EV Terminal. In this period of time a total of **17** issues were uncovered.

Protocol Summary

Protocol Name	EV Terminal
Repository	https://github.com/Sageterminal/Sage-Contracts
Date	April 17th 2024 - April 21st 2024
Protocol Type	DEX and Token Standard

Findings Count

Severity	**Amount**
Medium	10
Low	7
Total Findings	**17**

Summary of Findings

ID	Title	Severity	Status
[M-01]	A "team" can block another "team" from receiving funds	Medium	Resolved
[M-02]	Discrepancy in fee retrieval	Medium	Resolved
[M-03]	Accrued fees from the buy() or sell() functions could be lost	Medium	Resolved
[M-04]	Loss of ETH if both owner and feeCollector renounce	Medium	Resolved
[M-05]	Liquidity Provider cannot add liquidity after removing liquidity	Medium	Resolved
[M-06]	Factory owner can prevent fee collector from claiming fees	Medium	Resolved
[M-07]	The maxWallet variable can be bypassed	Medium	Acknowledged
[M-08]	Missing deadline on the swap functions	Medium	Acknowledged
[M-09]	Malicious liquidity provider can rug-pull users	Medium	Acknowledged
[M-10]	The Owner should be able to change the Liquidity Provider	Medium	Acknowledged
[L-01]	Function disableFee() can corrupt return value of fee()	Low	Resolved
[L-02]	Incorrect value emitted in the event RemoveLiquidity()	Low	Resolved
[L-03]	Unused variable lastTransaction	Low	Resolved
[L-04]	Approval race conditions on the	Low	Resolved

	ERC314Implementation		
[L-05]	Fee is applied asymmetrically in sell() comparing to buy()	Low	Acknowledged
[L-06]	WETH() and token0() are hardcoded	Low	Acknowledged
[L-07]	Malicious actor can inflate the token price	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] A "team" can block another "team" from receiving funds

Severity

Impact: High

Likelihood: Low

Description

Token deployments using the `ERC314Factory` accrue a fee for two teams:

1. `evFeeCollector`
2. `dzhvFeeCollector`

```
function claimFees() external {
        uint256 totalAccruedAmount = accruedFeeAmount;
        if (totalAccruedAmount > address(this).balance) {
            totalAccruedAmount = address(this).balance;
        }

        uint256 evTeamShare = (totalAccruedAmount * split) / 10000;
        uint256 dzhvTeamShare = totalAccruedAmount - evTeamShare;

        accruedFeeAmount = 0;

        (bool successEvFeeCollect, ) = payable
          (evFeeCollector).call{value: evTeamShare}("");
        require(successEvFeeCollect, "Transfer of EV team share failed");

        (bool successDzhvFeeCollect, ) = payable
          (dzhvFeeCollector).call{value: dzhvTeamShare}("");
        require(successDzhvFeeCollect, "Transfer of DZHV team share failed");
    }
```

There could be a state where one of the teams is a contract and:

1. Compromised

2. Malicious
3. Out of business / sanctioned

In the above states, the team could revert on transfers and prevent the other team from receiving their share of fees.

Recommendations

Consider not reverting on failure

[M-02] Discrepancy in fee retrieval

Severity

Impact: Medium

Likelihood: Medium

Description

When creating an `ERC314 token` using the `EVToken` contract, the fee amount to be charged in the `buy()/sell()` functions can be specified. Later, this fee will be directly stored in memory when initializing the contract in the `constructor`:

```
File: EvFactoryProxyOptFee.sol
555:
556: contract EVToken is ERC314Implementation {
557:     constructor(
558:         address implementation,
559:         address factory,
560:         address owner,
561:         uint totalSupply,
562:         uint fee,
563:         string memory name,
564:         string memory symbol
565:     ) {
566:         require(fee <= 500, "max 5% fee");
...
...
610:
611:         uint8 feeBytesNeeded = bytesNeeded(fee);
612:         assembly { mstore8(sizeCursor, feeBytesNeeded) }
613:         sizeCursor += 0x1;
614:         assembly { mstore(byteCursor, shl(mul(0x8, sub
  (0x20, feeBytesNeeded)), fee)) }
615:         byteCursor += feeBytesNeeded;
616:         assembly { mstore(0x40, add(mload(0x40), add
  (0x1, feeBytesNeeded))) }
...
...
634: }
```

Then, for example, in the `ERC314Implementation::buy()` function, the fee is retrieved from the `_opt.fee` variable (`EvFactoryProxyOptFee#L446`) to calculate the amount to charge:

```
File: EvFactoryProxyOptFee.sol
443:     function buy(uint256 amountOutMin) public payable {
...
...
446:         uint256 feeAmount = (msg.value * _opt.fee) / 10000;
447:
448:         uint256 ETHafterFee;
449:         unchecked {
450:             ETHafterFee = msg.value - feeAmount;
451:         }
...
...
473:     }
```

The issue is that when initializing the `EVToken` contract with a fee, this fee is stored in memory; however, the `_opt.fee` variable remains empty. This creates a discrepancy because, on one hand, the `ERC314Implementation::fee()` function retrieves the `fee` amount specified during the `EVToken` contract initialization, but on the other hand, the `_opt.fee` variable remains empty.

As seen, the `fee()` function first retrieves the value from the `_opt.fee` variable, and if it is empty, it retrieves it from memory.

```
File: EvFactoryProxyOptFee.sol
    function fee() public view virtual returns (uint16) {
        if (_opt.feeDisable) return 0;
        if (_opt.fee != 0) return _opt.fee;
        assembly {
            extcodecopy(address(), 0x20, 0x55, 0x4)
            let lengths := mload(0x20)
            let offset := add(0x55, 0x4)
            offset := add(offset, byte(0x0, lengths))
            let length := byte(0x1, lengths)
            extcodecopy(address(), sub(0x20, length), offset, length)
            return(0x0, 0x20)
        }
    }
```

The following test shows how the `EVToken` is initialized with a `fee=100`, then asserts to verify that the `fee()` function has a value of `100`. Later, in step 3, when Alice makes a purchase, the `accruedFeeAmount` remains zero:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import {BlankERC314, EVToken} from "../Sage-Contracts/EvFactoryProxyOptFee.sol";

contract EvToken is Test {

    EVToken public evToken;
    address public owner = address(this);
    address public alice = address(0x41123);

    function setUp() public {
        BlankERC314 implementation = new BlankERC314();
        evToken = new EVToken(
            address(implementation),
            address(this),
            owner,
            100e18, // totalSupply
            100, // fee
            "TestEvToken",
            "TEV"
        );
        evToken.initialize(10);  // 10% percent to deployer
    }

    function test_evTokenFeeDiscrepancy() public {
        //
        // 1. Assert fees are set to `100`
        assertEq(evToken.fee(), 100);
        //
        // 2. Owner add liquidity in order to open trading
        vm.deal(owner, 100 ether);
        uint32 timeTillUnlockLiquidity = uint32(block.timestamp + 12);
        evToken.addLiquidity{value: 10 ether}(timeTillUnlockLiquidity);
        //
        // 3. Alice buys `evTokens`. `accruedFeeAmount` should be incremented
        // but the
        // fees are zero
        vm.deal(alice, 100 ether);
        vm.prank(alice);
        evToken.buy{value: 1 ether}(0);
        assertEq(evToken.accruedFeeAmount(), 0);
    }
}
```

This proves that when initializing the `EVToken` with a `non-zero fee value`, it will not be charged immediately. You must manually call the `ERC314Implementation::setTradingFee` function with a new `fee amount` to update `_opt.fee` and have it charged in the `buy()/sell()` functions, which is redundant:

```
File: EvFactoryProxyOptFee.sol
    function setTradingFee(uint16 _fee) external onlyOwner {
        require(_fee <= 500, "max 5% fee");
        _opt.fee = _fee;
        if (_fee == 0) _opt.feeDisable = true;
        if (_fee != 0) _opt.feeDisable = false;
    }
```

Recommendations

The recommendation is to initialize `_opt.fee` in `ERC314Implementation::initialize`

[M-03] Accrued fees from the `buy()` or `sell()` functions could be lost

Severity

Impact: High

Likelihood: Low

Description

The `feeCollector` can renounce fees using the `ERC314Implementation::setFeeCollector` function, setting the variable `_feeCollector==address(0)`:

```
File: EvFactoryProxyOptFee.sol
    function setFeeCollector(address _newFeeCollector) external onlyOwner {
        _feeCollector = _newFeeCollector;
        if (_newFeeCollector == address(0)) _opt.feeCollectorRenounced = true;
        if (_newFeeCollector != address(0)) _opt.feeCollectorRenounced = false;
    }
```

The issue arises because fees generated in the `buy()` or `sell()` functions continue to accumulate even after the `feeCollector` has renounced them. If someone calls the `claimFees()` function (publicly accessible), then these accumulated fees will be sent to `address(0)` (`EvFactoryProxyOptFee#L492`), causing them to be lost:

```
File: EvFactoryProxyOptFee.sol
    function claimFees() external {
        uint256 totalAccruedAmount = accruedFeeAmount;
        if (totalAccruedAmount > address(this).balance) {
            totalAccruedAmount = address(this).balance;
        }

        uint256 factoryShare =
//(totalAccruedAmount * 10) / 100; // 10% to factory owner
        uint256 ownerShare = totalAccruedAmount - factoryShare;

        accruedFeeAmount = 0;

        if(factoryShare > 0) {
            (bool successFactory, ) = payable(IFactory(this.factory
  ()).getOwner()).call{value: factoryShare}("");
            require(successFactory, "Transfer of factory share failed");
        }

        (bool successOwner, ) = payable(this.feeCollector
  ()).call{value: ownerShare}("");
        require(successOwner, "Transfer of owner share failed");
    }
```

The following test demonstrates how the `feeCollector` renounces fees (`feeCollector==address(0)`), fees continue to accumulate as `Alice` executes the `buy()` function, but then a `malicious user` calls the `claimFees()` function, causing the `ETH` accumulated in the `accruedFeeAmount` variable to be sent to `address(0)` resulting in the loss of those fees:

```solidity
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import {BlankERC314, EVToken} from "../Sage-Contracts/EvFactoryProxyOptFee.sol";

contract EvToken is Test {

    EVToken public evToken;
    address public owner = address(0x02335);
    address public alice = address(0x41123);

    function setUp() public {
        BlankERC314 implementation = new BlankERC314();
        evToken = new EVToken(
            address(implementation),
            address(this), // factory
            owner,
            100e18, // totalSupply
            100, // fee
            "TestEvToken",
            "TEV"
        );
        evToken.initialize(10);  // 10% percent to deployer
    }

    // address(this) is the `factory`, so it needs the `getOwner` function
    function getOwner() external pure returns(address) { return address
      (0x03435); }

    function test_lostFeesWhenFeeCollectorRenounces() public {
        vm.prank(owner);
        evToken.setTradingFee(100);
        //
        // 1. Owner add liquidity in order to open trading
        vm.deal(owner, 100 ether);
        vm.deal(alice, 100 ether);
        uint32 timeTillUnlockLiquidity = uint32(block.timestamp + 12);
        vm.prank(owner);
        evToken.addLiquidity{value: 10 ether}(timeTillUnlockLiquidity);
        //
        // 2. Alice buys `evTokens` in order to increment `accruedFeeAmount`
        vm.prank(alice);
        evToken.buy{value: 1 ether}(0);
        uint256 accruedFeesFirstBuy = evToken.accruedFeeAmount();
        assertGt(accruedFeesFirstBuy, 0);
        //
        // 3. Owner renounce fees and feeCollector is address(0)
        vm.prank(owner);
        evToken.setFeeCollector(address(0));
        assertEq(evToken.feeCollector(), address(0));
        assertGt(evToken.accruedFeeAmount
        //(), 0); // accruedFeeAmount still positive
        //
        // 4. Alice buys more and the fees are accrued even more
        //(`accruedFeeAmount` is incremented)
        evToken.buy{value: 1 ether}(0);
        assertGt(evToken.accruedFeeAmount
        //(), accruedFeesFirstBuy);  // new accruedFeeAmount > old accruedFeeAmound
        //

        uint256 ethBalanceBefore = address(evToken).balance;
        evToken.claimFees();
        assertGt(ethBalanceBefore, address
        //(evToken).balance); // ethBalanceBefore > currentEthBalance
    }
}
```

Recommendations

Limit the use of `claimFees()` only to the `feeCollector` or the `factory.owner()`.

[M-04] Loss of `ETH` if both `owner` and `feeCollector` renounce

Severity

Impact: High

Likelihood: Low

Description

The `feeCollector` can renounce using the function `setFeeCollector(address(0))`:

```
File: EvFactoryProxyOptFee.sol
    function setFeeCollector(address _newFeeCollector) external onlyOwner {
        _feeCollector = _newFeeCollector;
        if (_newFeeCollector == address(0)) _opt.feeCollectorRenounced = true;
        if (_newFeeCollector != address(0)) _opt.feeCollectorRenounced = false;
    }
```

Similarly, the `owner` can also renounce with the function:

```
File: EvFactoryProxyOptFee.sol
    function renounceOwnership() external onlyOwner {
        if (!_opt.feeCollectorRenounced && _feeCollector == address(0)) _feeCollector = this.owner();
        if (!_opt.liquidityProviderRenounced && _liquidityProvider == address(0)) _liquidityPro
        _opt.ownerRenounced = true;
    }
```

However, this behavior can lead to some issues, as if both renounce their roles, fees will continue to be collected within the `buy()` and `sell()` functions, but within the `claimFees()` function, a percentage is still calculated for the `feeCollector`, even though the `feeCollector` is permanently set to `address(0)` as the owner has also renounced.

```
File: EvFactoryProxyOptFee.sol
    function claimFees() external {
...
...
        uint256 factoryShare =
//(totalAccruedAmount * 10) / 100; // 10% to factory owner
        uint256 ownerShare = totalAccruedAmount - factoryShare;
...
...
    }
```

The following test demonstrates how fees continue to accrue even when both `feeCollector` and `owner` have renounced their roles, causing the `claimFees()` function to calculate a percentage for the `feeCollector`, which is permanently set to `address(0)`, resulting in the loss of that percentage.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import {BlankERC314, EVToken} from "../Sage-Contracts/EvFactoryProxyOptFee.sol";

contract EvToken is Test {

    EVToken public evToken;
    address public owner = address(0x02335);
    address public alice = address(0x41123);

    function setUp() public {
        BlankERC314 implementation = new BlankERC314();
        evToken = new EVToken(
            address(implementation),
            address(this), // factory
            owner,
            100e18, // totalSupply
            100, // fee
            "TestEvToken",
            "TEV"
        );
        evToken.initialize(10);  // 10% percent to deployer
    }

    // this is the `factory`, so it needs the getOwner function
    function getOwner() external pure returns(address) { return address
      (0x03435); }

    function test_feesAreAccruedEvenWhenOwnerRenounces() public {
        vm.prank(owner);
        evToken.setTradingFee(100);
        //
        // 1. Owner add liquidity in order to open trading
        vm.deal(owner, 100 ether);
        vm.deal(alice, 100 ether);
        uint32 timeTillUnlockLiquidity = uint32(block.timestamp + 12);
        vm.prank(owner);
        evToken.addLiquidity{value: 10 ether}(timeTillUnlockLiquidity);
        //
        // 2. Alice buys `evTokens` in order to increment `accruedFeeAmount`
        vm.prank(alice);
        evToken.buy{value: 1 ether}(0);
        uint256 accruedFeesFirstBuy = evToken.accruedFeeAmount();
        assertGt(accruedFeesFirstBuy, 0);
        //
        // 3. Owner renounces to everything. feeCollector is address
        //(0) and owner() is zero
        vm.startPrank(owner);
        evToken.setFeeCollector(address(0));
        evToken.renounceOwnership();
        assertEq(evToken.feeCollector(), address(0));
        assertEq(evToken.owner(), address(0));
        assertGt(evToken.accruedFeeAmount
        //(), 0); // accruedFeeAmount still positive
        vm.stopPrank();
        //
        // 4. Alice buys more and the fees are accrued even more
        vm.prank(alice);
        evToken.buy{value: 1 ether}(0);
        assertGt(evToken.accruedFeeAmount
        //(), accruedFeesFirstBuy);  // new accruedFeeAmount > old accruedFeeAmound
        //

        uint256 ethBalanceBefore = address(evToken).balance;
        evToken.claimFees();
        assertEq(evToken.accruedFeeAmount(), 0);
    }
}
```

Recommendations

It is recommended that if both `feeCollector` and `owner` have renounced their roles, then 100% of the accrued fees should go to the `Factory.owner()`.

[M-05] Liquidity Provider cannot add liquidity after removing liquidity

Severity

Impact: Medium

Likelihood: Medium

Description

When liquidity is removed using `removeLiquidity` - `_opt.liquidityAdded` is not set to `false`. Therefore `addLiquidity` cannot be called again due to the check: `require(_opt.liquidityAdded == false, "Liquidity already added");`.

```
function addLiquidity(
        uint32 _timeTillUnlockLiquidity
    ) public payable onlyLiquidityProvider {
        require(_opt.liquidityAdded == false, "Liquidity already added");

        _opt.liquidityAdded = true;
-----------
    }

    function removeLiquidity() public onlyLiquidityProvider {
        require
          (block.timestamp > _opt.timeTillUnlockLiquidity, "Liquidity locked");
        _opt.tradingEnable = false;

        (uint256 reserveETH, ) = getReserves();

        (bool success, ) = payable(msg.sender).call{value: reserveETH}("");
        if (!success) {
            revert("Could not remove liquidity");
        }
        emit RemoveLiquidity(address(this).balance);
    }
```

Therefore, once the `LiquidityProvider` removes liquidity - it cannot be added again.

Consider a scenario where the owner is renounced and there is a need to disable trading and temporarily remove the liquidity of the contract. In such a case after removing the funds there is no way to add them back and enable trading again.

Recommendations

Add `_opt.liquidityAdded = false;` to `removeLiquidity`

[M-06] Factory owner can prevent fee collector from claiming fees

Severity

Impact: High

Likelihood: Low

Description

`claimFees` function is used to claim fees that accrue over time. 10 percent of the fees go to the factory owner.

If the factory owner is a contract that is compromised or malicious - he can DOS ***EVERY** deployed token from claiming fees simply by reverting the transaction.

```
function claimFees() external {
--------------------
        if(factoryShare > 0) {
            (bool successFactory, ) = payable(IFactory(this.factory()).getOwner
            //()).call{value: factoryShare}(""); // @audit malicious factory owner can
            require(successFactory, "Transfer of factory share failed");
        }

        (bool successOwner, ) = payable(this.feeCollector
          ()).call{value: ownerShare}("");
        require(successOwner, "Transfer of owner share failed");
    }
```

Recommendations

Consider not reverting if the `successFactory` value is false. Instead, consider keeping a separate accounting and a pull instead of push mechanism for the factory in case the push fails

[M-07] The `maxWallet` variable can be bypassed

Severity

Impact: Low

Likelihood: High

Description

The `maxWallet` variable helps determine the maximum number of `ERC` tokens a user can purchase. In the `ERC314Implementation::buy()` function, it is validated that the user does not exceed `maxWallet`.

```
File: EvFactoryProxyOptFee.sol
    function buy(uint256 amountOutMin) public payable {
...
...
        if (_opt.maxWalletEnable) {
            require(
                tokenAmount + _balances[msg.sender] <= maxWallet,
                "Max wallet exceeded"
            );
        }
...
...
    }
```

However, this restriction can be bypassed. The following test demonstrates how in the end, the `Alice2` wallet has a balance that exceeds `maxWallet`.

1. There is a `maxWallet` of `10e18` configured for the `ERC314`.
2. Alice purchases a quantity of tokens using the `buy()` function.
3. Alice attempts to purchase more; however, the transaction is reverted due to the error `Max wallet exceeded`.
4. Alice decides to transfer her balance to another controlled user, `Alice2`.
5. Alice can now purchase more `ERC314` tokens, and then transfer them to `Alice2`.

In the end, `Alice2` has a balance that exceeds the `maxWallet` restriction.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import {BlankERC314, EVToken} from "../Sage-Contracts/EvFactoryProxyOptFee.sol";

contract EvToken is Test {

    EVToken public evToken;
    address public owner = address(this);
    address public alice = address(0x41123);

    function setUp() public {
        BlankERC314 implementation = new BlankERC314();
        evToken = new EVToken(
            address(implementation),
            address(this),
            owner,
            100e18, // totalSupply
            100, // fee
            "TestEvToken",
            "TEV"
        );
        evToken.initialize(10);  // 10% percent to deployer
    }

    function test_maxWalletBypass() public {
        assertEq(evToken.maxWallet(), 0);
        assertEq(evToken.maxWalletEnable(), false);
        //
        // 1. Activate maxWallet
        evToken.setMaxWallet(10e18);
        evToken.enableMaxWallet(true);
        assertEq(evToken.maxWallet(), 10e18);
        assertEq(evToken.maxWalletEnable(), true);
        // Owner add liquidity
        vm.deal(owner, 100 ether);
        uint32 timeTillUnlockLiquidity = uint32(block.timestamp + 12);
        evToken.addLiquidity{value: 10 ether}(timeTillUnlockLiquidity);
        //
        // 2. Alice buys `evTokens`
        vm.deal(alice, 100 ether);
        vm.startPrank(alice);
        evToken.buy{value: 1 ether}(0);
        //
        // 3. Alice buy more tokens but the transaction is reverted by wallet
        // exceeded error
        vm.expectRevert("Max wallet exceeded");
        evToken.buy{value: 1 ether}(0);
        //
        // 4. Alice transfer this balance to another controlled address
        address alice2 = address(0x411232);
        evToken.transfer(alice2, evToken.balanceOf(alice));
        //
        // 5. Alice now can buy more `evTokens`. She transfers again to alice2
        // Now `alice2` has more tokens than the `maxWallet`
        evToken.buy{value: 1 ether}(0);
        evToken.transfer(alice2, evToken.balanceOf(alice));
        assertGt(evToken.balanceOf(alice2), evToken.maxWallet
        //());  // alice2.balance > maxWallet
    }
}
```

Recommendations

It is recommended to verify the `maxWallet` restriction in transfer functions.

EV Terminal response

Our intention for this is to prevent people from buying all the supply at once in 1 tx. They are okay to buy over multiple transactions/ multi wallets.

[M-08] Missing deadline on the swap functions

Severity

Impact: Low

Likelihood: High

Description

Swap functions don't have a deadline parameter. This parameter can provide the user an option to limit the execution of their pending transaction. Without a deadline parameter, users can execute their transactions at unexpected times when market conditions are unfavorable.

However, this is not a big problem in this case because the functions have slippage protection. Even though the users will get at least as much as they set, they may still be missing out on positive slippage if the exchange rate becomes favorable when the transaction is included in a block.

Recommendations

Introduce a deadline parameter in `buy()` and `sell()`

EV Terminal comment

Adding deadline logic would incur more gas. The purpose of the protocol is to reduce swap gas fees on ethereum and simplify the contracts. We acknowledge the issue and want to stick with just using slippage for protection.

[M-09] Malicious liquidity provider can rug-pull users

Severity

Impact: High

Likelihood: Low

Description

There are a few ways for malicious liquidity to rug-pull users:

Calling `removeLiquidity` without previously calling `addLiquidity`

A token owner can call `enableTrading` to allow trading without liquidity. After some time, when funds accumulated in the contract - a malicious liquidity provider can call `removeLiquidity` to send all the funds to himself.

Setting a low `timeTillUnlockLiquidity`

There is no restriction to a minimum `timeTillUnlockLiquidity`. Therefore a malicious liquidity provider can set `timeTillUnlockLiquidity` to a low value. The token would seem open for trading and already funded because `_opt.liquidityAdded == true` however the liquidity provider can call `removeLiquidity` to remove the funds.

Not extending `timeTillUnlockLiquidity`.

Consider a token that a liquidity provider has locked funds for a long time (24 months). After the 24 months are over - the liquidity provider can take ***ALL*** the funds of the contract and not just his supplied liquidity

Recommendations

1. When trading also check `_opt.liquidityAdded == true`
2. In `initialize` function add a minimum `timeTillUnlockLiquidity` value. This should be enforced in `addLiquidity`

3. Allow the liquidity provider to remove only his initial provided liquidity and not the entire ETH reserve of the contract.

EV Terminal comment

Other dexes also have the same design, we want to give them the flexibility. Many projects out there don't lock their liquidity and keep on multisig or just leave it unlocked in case of a potential migration.

[M-10] The Owner should be able to change the Liquidity Provider

Severity

Impact: High

Likelihood: Low

Description

Liquidity providers have the ability to remove reserves which will drain the contracts balance. Only tokens with a trusted liquidity provider are likely to be active - if the liquidity provider account is compromised:

- Private key leaked
- Seized

In such cases, the `owner` should be able to replace the liquidity provider before the liquidity lock is reached. However, the `owner` cannot replace the liquidity provider because `setLiquidityProvider` can only be called by the liquidity provider

```
function setLiquidityProvider(
        address _newLiquidityProvider
    ) external onlyLiquidityProvider {
        _liquidityProvider = _newLiquidityProvider;
        if (_newLiquidityProvider == address
          (0)) _opt.liquidityProviderRenounced = true;
        if (_newLiquidityProvider != address
          (0)) _opt.liquidityProviderRenounced = false;
    }
```

Recommendations

Consider changing the `onlyLiquidityProvider` to allow calls from the liquidity provider AND the owner.

EV Terminal comment

We want to keep the design so that even when owner is renounced, liquidity can still be extended/withdrawn if needed.

8.2. Low Findings

[L-01] Function `disableFee()` can corrupt return value of `fee()`

For some reason `disableFee()` can set `_opt.feeDisable = true`:

```
function disableFee(bool _disable) external onlyOwner {
        _opt.feeDisable = _disable;
        _opt.fee = 0;
    }
```

It will result in a default value instead of 0 returned from `fee()` because `_opt.fee == 0` and `_opt.feeDisable == true`:

```
function fee() public view virtual returns (uint16) {
        if (_opt.feeDisable) return 0;
        if (_opt.fee != 0) return _opt.fee;
        assembly {
            extcodecopy(address(), 0x20, 0x55, 0x4)
            let lengths := mload(0x20)
            let offset := add(0x55, 0x4)
            offset := add(offset, byte(0x0, lengths))
            let length := byte(0x1, lengths)
            extcodecopy(address(), sub(0x20, length), offset, length)
            return(0x0, 0x20)
        }
    }
```

Consider removing function `disableFee()` because it contains the same logic as `setTradingFee()`

[L-02] Incorrect value emitted in the event `RemoveLiquidity()`

Ensure that the correct value is emitted, because currently, it emits an amount of ETH fees in the contract

```
function removeLiquidity() public onlyLiquidityProvider {
        require
          (block.timestamp > _opt.timeTillUnlockLiquidity, "Liquidity locked");

        _opt.tradingEnable = false;

        (uint256 reserveETH, ) = getReserves();

        (bool success, ) = payable(msg.sender).call{value: reserveETH}("");
        if (!success) {
            revert("Could not remove liquidity");
        }

@>      emit RemoveLiquidity(address(this).balance);
    }
```

[L-03] Unused variable `lastTransaction`

Contract `ERC314Implementation` contains storage variable `lastTransaction` which is never used. Consider removing it or adding missing functionality.

[L-04] Approval race conditions on the `ERC314Implementation`

Description

The "approve race condition" in `ERC314Implementation::approve` is a vulnerability that occurs when changing an allowance from one non-zero value to another. If an owner first approves a spender for a certain amount of tokens and then decides to change this approval, there is a risk that the spender can execute a `transferFrom` to move the originally approved amount if they act during the window between the submission and confirmation of the new approval transaction. This can allow the spender to transfer more tokens than the owner intended.

The following test demonstrates the vulnerability:

1. Owner approves Alice to spend `5e18` of his tokens.
2. Owner decides to reduce Alice's allowance to `2e18` tokens and sends this new approval transaction.
3. Before Owner's new approval transaction is confirmed, Alice notices the pending transaction and quickly transfers the `5e18` tokens.

4. Owner's transaction to reduce the allowance to `2e18` tokens confirms. Alice can transfer this new allowance. Finally, Alice obtains more tokens than intended by the Owner.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import {BlankERC314, EVToken} from "../Sage-Contracts/EvFactoryProxyOptFee.sol";

contract EvToken is Test {

    EVToken public evToken;
    address public owner = address(this);
    address public alice = address(0x41123);

    function setUp() public {
        BlankERC314 implementation = new BlankERC314();
        evToken = new EVToken(
            address(implementation),
            address(this),
            owner,
            100e18, // totalSupply
            100, // fee
            "TestEvToken",
            "TEV"
        );
        evToken.initialize(10);  // 10% percent to deployer
    }

    function test_approveRace() public {
        //
        // 1. Owner approves 5e18 to Alice. Assert alice's allowance
        evToken.approve(alice, 5e18);
        assertEq(evToken.allowance(owner, alice), 5e18);
        //

        vm.prank(alice);
        evToken.transferFrom(owner, alice, 5e18);
        assertEq(evToken.allowance(owner, alice), 0);
        //
        // 3. Owner update alice's allowance is executed to 2e18
        evToken.approve(alice, 2e18);
        assertEq(evToken.allowance(owner, alice), 2e18);
        //
        // 4. Alice transfer again and she owns 7e18 tokens.
        vm.prank(alice);
        evToken.transferFrom(owner, alice, 2e18);
        assertEq(evToken.balanceOf(alice), 7e18);
    }
}
```

We understand that there is no definitive solution to this problem; however, the following proposals can be considered:

- **Use the `increaseAllowance` and `decreaseAllowance` methods**: These functions change the allowance in a way that can help prevent this vulnerability by adding or subtracting from the current allowance rather than setting it to a new value outright.

- **Set the allowance to zero before setting it to a new value**: If changing the allowance is necessary, the owner should first set the allowance to zero and then set it to the new desired value in two separate transactions. This two-step process closes the window for exploiting the race condition.

[L-05] Fee is applied asymmetrically in `sell()` comparing to `buy()`

The fee is always taken in ETH, so in `buy()` it firstly subtracts the fee from the input amount and only after performing a swap:

```
function buy(uint256 amountOutMin) public payable {
        require(_opt.tradingEnable, "Trading not enable");
        uint256 feeAmount = (msg.value * _opt.fee) / 10000;

        uint256 ETHafterFee;
        unchecked {
@>          ETHafterFee = msg.value - feeAmount;
        }

        unchecked {
            accruedFeeAmount += feeAmount;
        }
        (uint256 reserveETH, uint256 reserveToken) = getReserves();

@>      uint256 tokenAmount = (ETHafterFee * reserveToken) / reserveETH;
        ...
    }
```

However, in `sell()` fee is taken on the output amount. It means that fee size is different from the same amount in functions `buy()` and `sell()` because in `sell()` fee is taken after slippage. So basically fee is always greater in `buy()` than `sell()` because in buy slippage is not applied to the fee.

EV Terminal comment:

For simplicity of the calculation we'll keep the design this way and accept that slippage will affect the discrepancy in fees.

[L-06] `WETH()` and `token0()` are hardcoded

`WETH()` and `token0()` are hardcoded to point to `0x4200000000000000000000000000000000000006` which is the WETH address on optimism chain.

```
function WETH() public pure virtual returns (address) {
        return address(0x4200000000000000000000000000000000000006);
    }

    function token0() public pure virtual returns (address) {
        return address(0x4200000000000000000000000000000000000006);
    }
```

However, if this contract is intended to be deployed on other chains it could cause unexpected/wanted behavior. Consider updating these values as part of the `initialize` function

EV Terminal comment

We will only deploy on ETH and want to keep the contract simple thus hardcoding.

[L-07] Malicious actor can inflate the token price

When trading, the amount of tokens to receive when buying/selling follows the `XY=K` formula and is related to the ratio between the contract's `ETH` reserve and the `token` reserve.

```
function getAmountOut(
        uint256 value,
        bool _buyBool
    ) public view returns (uint256) {
        (uint256 reserveETH, uint256 reserveToken) = getReserves();

        if (_buyBool) {
            uint256 valueAfterFee = (value * (10000 - _opt.fee)) / 10000;
            return ((valueAfterFee * reserveToken)) /
              (reserveETH + valueAfterFee);
        } else {
            uint256 ethValue = ((value * reserveETH)) / (reserveToken + value);
            ethValue = (ethValue * (10000 - _opt.fee)) / 10000;
            return ethValue;
        }
    }
```

However, `getReserves()` returns the `ETH` ***BALANCE*** of the contract and not just the liquidity provided by the liquidity provider and the funds accrued through `buy` and `sell` :

```
function getReserves() public view returns (uint256, uint256) {
        return (
            (address(this).balance - accruedFeeAmount),
            _balances[address(this)]
        );
    }
```

In normal `buy` operations `reserveToken` is reduced and `reserveETH` is increased proportionately and in accordance with `XY=K`.

A malicious actor can donate `ETH` which will ***only*** increase `reserveETH` and therefore unbalance the formula. This will artificially inflate the price of the token (making it cheaper).

Users that want to `sell` will receive less ETH than expected.

Consider in `getReserves` - Instead of using the `ETH` balance of the contract. Return for `reserveETH`:

- locked liquidity
- liquidity gained through "buy" minus liquidity sold through `sell`

This will promise a "controlled" and expected formula.

EV Terminal comment

Since there is very little incentive to do it and impact is limited we will keep the code and acknowledge it.