CD SECURITY

AUDIT REPORT

Whirl Protocol
February 2024

Introduction

A time-boxed security review of the **Whirl** protocol was done by **CD Security**, 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 **Whirl**

Whirl aims to guarantee its users the privacy of the transactions made through the protocol. Aside from that, it has its own ERC20 token - `WhirlToken` that can be vested. The protocols relies on a virtual layer as an additional privacy layer which was not in scope for this audit. The full technical overview and extended docs can be found here.

Threat Model

Privileged Roles & Actors

Security Interview

Q: What in the protocol has value in the market?

A: All the funds deposited as well as the Whirl's own token.

Q: In what case can the protocol/users lose money?

A: If the protocol deposited funds are drained or if the claimable tokens are wrongly distributed.

Q: What are some ways that an attacker achieves his goals?

A: To withdraw funds that are deposited by other users or steal their rewards for vested tokens.

Severity classification

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

Severity	Impact: High	Impact: Medium	Impact: 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

Security Assessment Summary

***review commit hash* - 51f20f0312d82f678130294e73e538f3de3046dd**

Scope

The following smart contracts were in scope of the audit:

- `contracts/Pausable.sol`
- `contracts/WhirlDeposit.sol`
- `contracts/WhirlRevDeposit.sol`
- `contracts/WhirlToken.sol`
- `contracts/WhirlVesting.sol`
- `zksync/WhirlDeposit.sol`

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

- Critical & High: 2 issues
- Medium: 2 issues
- Informational: 4 issues

Findings Summary

ID	Title	Severity
[H-01]	Lack of slippage control can lead to sandwich attacks	High
[H-02]	Hardcoded address of `UniswapV2Router02` will lead to problems on some chains	High
[M-01]	Deadline check is not effective	Medium
[M-02]	`claim` method should be usable while the `WhirlVesting` contract is paused	Medium
[I-01]	Missing event emissions in state changing methods	Informational

ID	Title	Severity
[I-02]	Use newer Solidity version with a stable pragma statement	Informational
[I-03]	Unused code	Informational
[I-04]	NatSpec docs are missing	Informational

Detailed Findings

[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 amount_) private {
        IUniswapV2Router cachedRouter = uniswapV2Router;

        address[] memory path = new address[](2);
        path[0] = address(this);
        path[1] = cachedRouter.WETH();

        _approve(address(this), address(cachedRouter), amount_);

        cachedRouter.swapExactTokensForETHSupportingFeeOnTransferTokens(
            amount_,
            0, //@audit hardcoded
            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,
            msg.sender,
            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] Hardcoded address of `UniswapV2Router02` will lead to problems on some chains

Severity

Impact: High, as calls to the router will fail

Likelihood: Medium, because there is no way to change the address and a redeployment will be needed

Description

Currently, the `UniswapV2Router02` address in `WhirlToken` is hardcoded as follows:

```
    address private constant _UNIV2_ROUTER
        = 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D;
```

This is correct for the ETH mainnet but the protocol is intended to be multi-chain and it will be deployed to many different networks as can be seen from the documentation. The problem is that the above address on BASE , for example, is an EOA and the Uniswap Router address is different. In reality, the router is deployed at different addresses on many chains. The address is hardcoded as a `constant` and cannot be changed later on.

This can also be true for the `marketWallet`, `revWallet`, and the `teamWallet`. Even though there are functionalities to change the wallets, it can take some time before the team realizes funds are sent to wrong addresses on specific chains.

Recommendations

Instead of hardcoding addresses, pass them as an input parameter in the constructor and check on every chain that you plan to deploy, which is the correct `UniswapV2Router02` address.

[M-01] Deadline check is not effective

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: Low, because a really low gas fee is needed to leave the transaction in the mempool for that long.

The deadline parameter in `swapExactTokensForETHSupportingFeeOnTransferTokens()` and `addLiquidityETH()` which are called in `_swapTokensForEth()` and `_addLiquidity()` is hardcoded to `block.timestamp`.

Example in `_addLiquidity()`:

```
uniswapV2Router.addLiquidityETH{value: ethAmount}(
            address(this),
            tokenAmount,
            0,
            0,
            msg.sender,
            block.timestamp //@audit deadline param is hardcoded
        );
```

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.

Let's take a look at a modifier that 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 chooses 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`.

[M-02] `claim` method should be usable while the `WhirlVesting` contract is paused

Severity

Impact: Medium, the users of the protocol won't be able to claim their vesting tokens

Likelihood: Medium, because it requires a malicious or compromised owner who will pause the contract and never unpause it

Description

The `WhirlVesting:claim` function is only callable when the `WhirlVesting` contract is not paused. This prevents a user from collecting accumulated funds. The admin of the `WhirlVesting` contract can potentially pause the contracts at any time, locking users out of their honestly earned funds.

```
function claim() external {
        _checkPaused();
        Vesting storage vesting = vestings[msg.sender];

        uint256 claimable = _claimable(vesting.total, vesting.claimed);

        if (claimable == 0) _revert(NothingToClaim.selector);

        unchecked {
            vesting.claimed += claimable;
        }
```

```
        _safeTransfer(token, msg.sender, claimable);

        emit Claimed(msg.sender, claimable);
}
```

Recommendations

Consider removing the `_checkPaused` from the `claim` function to allow claiming while the `WhirlVesting` contract is paused.

[I-01] 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:

- `startNow()`
- `updateSwapTokensAtAmount()`
- `updateMaxSwapTokens()`
- `updateMaxTradingAmount()`
- `updateMaxWalletAmount()`
- `updateFees()`
- `updateStageFees()`
- `updateFeeDistribution()`
- `renounceBlacklist()`

[I-02] Use newer Solidity version with a stable pragma statement

Currently, the contracts are using floatable `0.8.9` pragma. Consider using the same stable pragma version, as well as a newer version for all of the contracts inside the codebase. Also, as you plan to deploy the contracts to any other blockchains, aside from ETH main net, consider using the `0.8.19` as the `PUSH0` which is used in `0.8.20` is still not supported on some other L2 blockchains.

[I-03] Unused code

The following two variables are not used anywhere in the codebase. Either use or delete them:

- `error TransferFailed();`
- `event RevDeposit(address indexed user, uint256 amount);`

[I-04] NatSpec docs are missing

`@notice`, `@param` and `@return` fields are missing completely from all of the contracts in scope. For completeness and readability, it is recommended to document the NatSpec for all the functions in the contracts where feasible. NatSpec documentation is essential for better understanding of the code by developers and auditors and is strongly recommended. Please refer to the NatSpec format and follow the guidelines outlined there.