



**SHERLOCK**

# **SHERLOCK SECURITY REVIEW FOR**



**SHERLOCK**

**Prepared for:**

**dodo**

**Prepared by:**

**Sherlock**

**Lead Security Expert:**

**ak1**

**Dates Audited:**

**November 9 - November 12, 2022**

**Prepared on:**

**November 23, 2022**

## Introduction

DODO is a decentralized trading platform that uses the innovative Proactive Market Maker (PMM) algorithm to provide efficient on-chain liquidity for Web3 assets.

## Scope

The following contracts in the DODOEX/dodo-route-contract repo are in scope.

- /SmartRoute/DODORouteProxy.sol
- DODOApprove.sol
- DODOApproveProxy.sol
- IDODOApprove.sol
- lib/DecimalMath.sol
- lib/UniversalERC20.sol

## Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

## Issues found

Medium	High
5	0

## Issues not fixed or acknowledged

Medium	High
0	0

## Security experts who found valid issues

TrungOre  
ctf\_sec  
Tomo

jayphbee  
yixxas  
0x4non

sach1r0  
Nyx  
pashov



8olidity  
rvierdiiev  
Bnke0x0

0xNazgul  
EIKu  
virtualfact

defsec  
ak1



## Issue M-1: Use `safeTransferFrom()` instead of `transferFrom()`.

Source: <https://github.com/sherlock-audit/2022-11-dodo-judging/issues/47>

### Found by

yixxas, 0x4non, sach1r0, Tomo, Nyx

### Summary

The `ERC20.transfer()` and `ERC20.transferFrom()` functions return a boolean value indicating success. This parameter needs to be checked for success. Some tokens do not revert if the transfer failed but return false instead.

### Vulnerability Detail

Some tokens (like USDT) don't correctly implement the EIP20 standard and their `transfer/ transferFrom` function return void instead of a success boolean. Calling these functions with the correct EIP20 function signatures will always revert.

### Impact

Tokens that don't actually perform the transfer and return false are still counted as a correct transfer and tokens that don't correctly implement the latest EIP20 spec, like USDT, will be unusable in the protocol as they revert the transaction because of the missing return value.

### Code Snippet

<https://github.com/sherlock-audit/2022-11-dodo/blob/main/contracts/SmartRoute/DOORouteProxy.sol#L420>

<https://github.com/sherlock-audit/2022-11-dodo/blob/main/contracts/SmartRoute/DOORouteProxy.sol#L423>

### Tool used

Manual Review

### Recommendation

Recommend using OpenZeppelin's SafeERC20 versions with the `safeTransfer` and `safeTransferFrom` functions that handle the return value check as well as non-standard-compliant tokens.



## Discussion

### Evert0x

We think a medium is still valid, although no direct loss of funds, a failed token transfer should be caught.



## Issue M-2: Rounding error when call function `dodoMultiSwap()` can lead to revert of transaction or fund of user

Source: <https://github.com/sherlock-audit/2022-11-dodo-judging/issues/45>

### Found by

TrungOre

### Summary

The calculation of the proportion when do the split swap in function `_multiSwap` doesn't care about the rounding error

### Vulnerability Detail

The amount of `midToken` will be transferred to the each adapter can be calculated by formula `curAmount=curTotalAmount*weight/totalWeight`

It will lead to some scenarios when `curTotalAmount*curPoolInfo.weight` is not divisible by `curTotalWeight`, there will be some token left after the swap.

For some tx, if user set a `minReturnAmount` strictly, it may incur the reversion. For some token with small decimal and high value, it can make a big loss for the sender.

### Impact

- Revert the transaction because not enough amount of `toToken`
- Sender can lose a small amount of tokens

### Code Snippet

<https://github.com/sherlock-audit/2022-11-dodo/blob/main/contracts/SmartRoute/DODORouteProxy.sol#L415-L425>

### Tool used

Manual review

### Recommendation

Add a accumulation variable to maintain the total amount is transferred after each split swap. In the last split swap, instead of calculating the `curAmount` by formula above, just take the remaining amount to swap.



## Issue M-3: universalApproveMax will not work for some tokens that don't support approve type(uint256).max amount.

Source: <https://github.com/sherlock-audit/2022-11-dodo-judging/issues/41>

### Found by

Tomo, jayphbee

### Summary

universalApproveMax will not work for some tokens that don't support approve type(uint256).max amount.

### Vulnerability Detail

There are tokens that doesn't support approve spender type(uint256).max amount. So the universalApproveMax will not work for some tokens like UNI or COMP who will revert when approve type(uint256).max amount.

### Impact

Tokens that don't support approve type(uint256).max amount could not be swapped by calling externalSwap function.

### Code Snippet

<https://github.com/sherlock-audit/2022-11-dodo/blob/main/contracts/SmartRoute/DODORouteProxy.sol#L181-L183>

```
if (approveTarget != address(0)) {
    IERC20(fromToken).universalApproveMax(approveTarget, fromTokenAmount);
}
```

<https://github.com/sherlock-audit/2022-11-dodo/blob/main/contracts/SmartRoute/UniversalERC20.sol#L36-L48>

```
function universalApproveMax(
    IERC20 token,
    address to,
    uint256 amount
) internal {
    uint256 allowance = token.allowance(address(this), to);
    if (allowance < amount) {
        if (allowance > 0) {
```



```
        token.safeApprove(to, 0);  
    }  
    token.safeApprove(to, type(uint256).max);  
}  
}
```

## Tool used

Manual Review

## Recommendation

I would suggest approve only the necessary amount of token to the approveTarget instead of the `type(uint256).max` amount.





## Issue M-4: Issue when handling native ETH trade and WETH trade in DODO RouterProxy#externalSwap

Source: <https://github.com/sherlock-audit/2022-11-dodo-judging/issues/20>

### Found by

ctf\_sec

### Summary

Lack of logic to wrap the native ETH to WETH in function externalSwap

### Vulnerability Detail

The function exeternalSwap can handle external swaps with 0x, 1inch and paraswap or other external resources.

```
function externalSwap(
    address fromToken,
    address toToken,
    address approveTarget,
    address swapTarget,
    uint256 fromTokenAmount,
    uint256 minReturnAmount,
    bytes memory feeData,
    bytes memory callDataConcat,
    uint256 deadLine
) external payable judgeExpired(deadLine) returns (uint256 receiveAmount) {
    require(isWhiteListedContract[swapTarget], "DODORouteProxy: Not Whitelist
    ↪ Contract");
    require(isApproveWhiteListedContract[approveTarget], "DODORouteProxy: Not
    ↪ Whitelist Approve Contract");

    // transfer in fromToken
    if (fromToken != _ETH_ADDRESS_) {
        // approve if needed
        if (approveTarget != address(0)) {
            IERC20(fromToken).universalApproveMax(approveTarget,
            ↪ fromTokenAmount);
        }

        IDODOApproveProxy(_DODO_APPROVE_PROXY_).claimTokens(
            fromToken,
            msg.sender,
            address(this),
```



```

        fromTokenAmount
    );
}

// swap
uint256 toTokenOriginBalance;
if(toToken != _ETH_ADDRESS_) {
    toTokenOriginBalance = IERC20(toToken).universalBalanceOf(address(this));
} else {
    toTokenOriginBalance = IERC20(_WETH_).universalBalanceOf(address(this));
}

```

note the code above, if the fromToken is set to `_ETH_ADDRESS_`, indicating the user wants to trade with native ETH pair. the function does has payable modifier and user can send ETH along when calling this function.

However, the toTokenOriginBalance is check the only *WETH* balance instead of ETH balance.

```

if(toToken != _ETH_ADDRESS_) {
    toTokenOriginBalance = IERC20(toToken).universalBalanceOf(address(this));
} else {
    toTokenOriginBalance = IERC20(_WETH_).universalBalanceOf(address(this));
}

```

Then we do the swap:

```

(bool success, bytes memory result) = swapTarget.call{
    value: fromToken == _ETH_ADDRESS_ ? fromTokenAmount : 0
}(callDataConcat);

```

If the fromToken is `_ETH_ADDRESS_`, we send the user supplied fromTokenAmount without verifying that the fromTokenAmount.

Finally, we use the before and after balance to get the amount with received.

```

// calculate toToken amount
if(toToken != _ETH_ADDRESS_) {
    receiveAmount = IERC20(toToken).universalBalanceOf(address(this)) - (
        toTokenOriginBalance
    );
} else {
    receiveAmount = IERC20(_WETH_).universalBalanceOf(address(this)) - (
        toTokenOriginBalance
    );
}

```



We are checking the WETH amount instead of ETH amount again.

**The issue is that some trades may settle the trade in native ETH, for example**

<https://developers.paraswap.network/smart-contracts>

we can look into the Paraswap contract

<https://etherscan.io/address/0xDEF171Fe48CF0115B1d80b88dc8eAB59176FEe57#writeProxyContract>

If we click the implementation contract and see the method swapOnUniswapV2Fork

<https://etherscan.io/address/0x4ff0dec5f9a763aa1e5c2a962aa6f4edfee4f9ea#code>

Code line 927 - 944, which calls the function

```
function swapOnUniswapV2Fork(
    address tokenIn,
    uint256 amountIn,
    uint256 amountOutMin,
    address weth,
    uint256[] calldata pools
)
    external
    payable
{
    _swap(
        tokenIn,
        amountIn,
        amountOutMin,
        weth,
        pools
    );
}
```

which calls:

```
function _swap(
    address tokenIn,
    uint256 amountIn,
    uint256 amountOutMin,
    address weth,
    uint256[] memory pools
)
    private
    returns (uint256 tokensBought)
{
    uint256 pairs = pools.length;
```



```

require(pairs != 0, "At least one pool required");

bool tokensBoughtEth;

if (tokenIn == ETH_IDENTIFIER) {
    require(amountIn == msg.value, "Incorrect msg.value");
    IWETH(weth).deposit{value: msg.value}();
    require(IWETH(weth).transfer(address(pools[0]), msg.value));
} else {
    require(msg.value == 0, "Incorrect msg.value");
    transferTokens(tokenIn, msg.sender, address(pools[0]), amountIn);
    tokensBoughtEth = weth != address(0);
}

tokensBought = amountIn;

for (uint256 i = 0; i < pairs; ++i) {
    uint256 p = pools[i];
    address pool = address(p);
    bool direction = p & DIRECTION_FLAG == 0;

    tokensBought = NewUniswapV2Lib.getAmountOut(
        tokensBought, pool, direction, p >> FEE_OFFSET
    );
    (uint256 amount0Out, uint256 amount1Out) = direction
        ? (uint256(0), tokensBought) : (tokensBought, uint256(0));
    IUniswapV2Pair(pool).swap(
        amount0Out,
        amount1Out,
        i + 1 == pairs
            ? (tokensBoughtEth ? address(this) : msg.sender)
            : address(pools[i + 1]),
        ""
    );
}

if (tokensBoughtEth) {
    IWETH(weth).withdraw(tokensBought);
    TransferHelper.safeTransferETH(msg.sender, tokensBought);
}

require(tokensBought >= amountOutMin, "UniswapV2Router:
↳ INSUFFICIENT_OUTPUT_AMOUNT");
}

```

as can clearly see, the code first receive ETH, wrap ETH to WETH, then instead end,



unwrap the WETH to ETH and then send the ETH back to complete the trade.

```
if (tokensBoughtEth) {
    IWETH(weth).withdraw(tokensBought);
    TransferHelper.safeTransferETH(msg.sender, tokensBought);
}
```

In DODORouterProxy.sol#ExternalSwap however, we are using WETH balance before and after to check the received amount,

but if we call swapOnUniswapV2Fork on Paraswap router, the balance change for WETH would be 0

because as we see above, the method on paraswap side wraps ETH to WETH but in the end unwraps WETH and sends ETH back.

There is also a lack of a method to wrap the ETH to WETH before the trade, making the ETH-related order not tradeable.

## Impact

A lot of methods that do not use WETH to settle the trade will not be callable.

## Code Snippet

<https://github.com/sherlock-audit/2022-11-dodo/blob/main/contracts/SmartRoute/DODORouterProxy.sol#L158-L230>

## Tool used

Manual Review

## Recommendation

We recommend the project change from

```
// swap
uint256 toTokenOriginBalance;
if(toToken != _ETH_ADDRESS_) {
    toTokenOriginBalance = IERC20(toToken).universalBalanceOf(address(this));
} else {
    toTokenOriginBalance = IERC20(_WETH_).universalBalanceOf(address(this));
}
```

```
// swap
uint256 toTokenOriginBalance;
if(toToken != _ETH_ADDRESS_) {
```



```

        toTokenOriginBalance = IERC20(toToken).universalBalanceOf(address(this));
    } else {
        toTokenOriginBalance =
        ↪ IERC20(_ETH_ADDRESS).universalBalanceOf(address(this));
    }

```

If we want to use WETH to do the balance check, we can help the user wrap the ETH to WETH by calling before do the balance check.

```

IWETH(_WETH_).deposit(receiveAmount);

```

If we want to use WETH as the reference to trade, we also need to approve external contract to spend our WETH.

We can add

```

if(fromToken == _ETH_ADDRESS) {
    IERC20(_WETH_).universalApproveMax(approveTarget, fromTokenAmount);
}

```

We also need to verify the fromTokenAmount for

```

(bool success, bytes memory result) = swapTarget.call{
    value: fromToken == _ETH_ADDRESS ? fromTokenAmount : 0
}(callDataConcat);

```

we can add the check:

```

require(msg.value == fromTokenAmount, "invalid ETH amount");

```

## Discussion

### Attens1423

In our api, we require toToken is WETH when constructing callData. We will add some notes here. Thanks for noticing

### Evert0x

Even tough the API is requiring WETH we still think it's a valid issue as the contract has a payable modifier.



## Issue M-5: `call()` should be used instead of `transfer()` on an address payable

Source: <https://github.com/sherlock-audit/2022-11-dodo-judging/issues/5>

### Found by

ak1, 8solidity, sach1r0, ElKu, 0xNazgul, virtualfact, yixxas, Bnke0x0, defsec, 0x4non, pashov, Tomo, rvierdiev, Nyx

### Summary

#### Vulnerability Detail

The `transfer()` and `send()` functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example, EIP 1884 broke several existing smart contracts due to a cost increase of the `SLOAD` instruction.

### Impact

The use of the deprecated `transfer()` function for an address will inevitably make the transaction fail when:

- The claimer smart contract does not implement a payable function.
- The claimer smart contract does implement a payable fallback which uses more than 2300 gas unit.
- The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through proxy, raising the call's gas usage above 2300.
- Additionally, using higher than 2300 gas might be mandatory for some multisig wallets.

### Code Snippet

```
DODORouteProxy.sol#L152 payable(routeFeeReceiver).transfer(restAmount); DO  
DORouteProxy.sol#L489 payable(msg.sender).transfer(receiveAmount); Universal  
ERC20.sol#L29 to.transfer(amount);
```

### Tool used

Manual Review



## Recommendation

Use `call()` instead of `transfer()`, but be sure to respect the CEI pattern and/or add re-entrancy guards, as several hacks already happened in the past due to this recommendation not being fully understood.

More info on; <https://swcregistry.io/docs/SWC-134>

