
Armors Labs
Linkswap
Smart Contract Audit

Linkswap Audit Summary

Project name : Linkswap Contract

Project address: None

Code URL : https://github.com/zk-linkswap/linkswap-contracts

Commit : 2fcfb870dcb530017fe117924fcfbe6061151cbd

Project target : Linkswap Contract Audit

Blockchain : zkLink Nova

Test result : PASSED

Audit Info

Audit NO : 0X202403310026

Audit Team : Armors Labs

Audit Proofreading: https://armors.io/#project-cases

Linkswap Audit

The Linkswap team asked us to review and audit their Linkswap contract. We looked at the code and now publish our results.

Here is our assessment and recommendations, in order of importance.

Document information

Name	Auditor	Version	Date
Linkswap Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2024-03-31

Audit results

Notices

```
The Factory Owner can set swap fee rate
The Factory Owner can set swap fee receipt address
```

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the Linkswap contract. The above should not be construed as investment advice.

Based on the widely recognized security status of the current underlying blockchain and smart contract, this audit report is valid for 3 months from the date of output.

Disclaimer

Armors Labs Reports is not and should not be regarded as an "approval" or "disapproval" of any particular project or team. These reports are not and should not be regarded as indicators of the economy or value of any "product" or "asset" created by any team. Armors do not cover testing or auditing the integration with external contract or services (such as Unicrypt, Uniswap, PancakeSwap etc'…)

Armors Labs Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology. Armors does not guarantee the safety or functionality of the technology agreed to be analyzed.

Armors Labs postulates that the information provided is not missing, tampered, deleted or hidden. If the information provided is missing, tampered, deleted, hidden or reflected in a way that is not consistent with the actual situation, Armors Labs shall not be responsible for the losses and adverse effects caused. Armors Labs Audits should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

Audited target file

file	md5
/interfaces/router/ISwapRouter01.sol	635cc4ed6521ac033e46b4959ef98dd3
/interfaces/router/ISwapRouter02.sol	6f31943edc5a230a69bce6ec3b1908fd
/interfaces/router/IWNativeToken.sol	199bf1e1e34c088c34d199f3e26dd4b4
/libraries/core/UQ112x112.sol	3cb134a55f4912bb438f3e7dc865feae
/libraries/router/SwapRouterLibrary.sol	0e0db255b7a9b2232441c5275634cac5
/libraries/router/TransferHelper.sol	fb06f7def90c1bc6c5732491df453957
/libraries/wtoken/WNativeToken.sol	9bec376931efd10675acf465159b7c1d
/interfaces/IERC20.sol	600bed11009ad8c127e339641460a43c
/interfaces/ISwapCallee.sol	c74707b7440d19dd4d919bce438f1f4d
/interfaces/ISwapERC20.sol	5f05f39473e2fb47572789277f64d973
/interfaces/ISwapFactory.sol	1a33dcbfe1303d3321e89403e9cac29f
/interfaces/ISwapPair.sol	91f8ca7177de88d307d97d9210e04265
/interfaces/ISwapPairBase.sol	f327fac64db7a3cf4861880a02e7cb57
/libraries/Math.sol	1e8a5534edefbb647806f279b5ce0039
/libraries/Panic.sol	4a9c35a563e77a7ce4bac9d9be3ccf13
/libraries/SafeCast.sol	6a44339a6a9b4f733db339c47a85a22d
/libraries/SafeMath.sol	c2c3a87198302e388ddf273a3a184199
/SwapERC20.sol	811274ec3dbf2fed83cb47edbfb86a31
/SwapFactory.sol	3a57267d93dc3ab52381d6f68a1c3ea5
/SwapPair.sol	f2f1e6bc48742232800d55ca469263df
/SwapRouter.sol	a059bc20e9d402969d427eb4e021a15b

Vulnerability analysis

Vulnerability distribution

vulnerability level	number
Critical severity	0
High severity	0
Medium severity	0
Low severity	0

Summary of audit results

Vulnerability	status
Re-Entrancy	safe
Arithmetic Over/Under Flows	safe
Unexpected Blockchain Currency	safe
Delegatecall	safe
Default Visibilities	safe
Entropy Illusion	safe
External Contract Referencing	safe
Short Address/Parameter Attack	safe
Unchecked CALL Return Values	safe
Race Conditions / Front Running	safe
Denial Of Service (DOS)	safe
Block Timestamp Manipulation	safe
Constructors with Care	safe
Unintialised Storage Pointers	safe
Floating Points and Numerical Precision	safe
tx.origin Authentication	safe
Permission restrictions	safe

Contract file

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.17;

// Router interface can be combined into one
```

```
// Here we divide it following UniSwap's style for readability
interface ISwapRouter01 {
    function factory() external view returns (address);

    function WNativeToken() external view returns (address);

    function addLiquidity(
        address tokenA,
        address tokenB,
        uint256 amountADesired,
        uint256 amountBDesired,
        uint256 amountAMin,
        uint256 amountBMin,
        address to,
        uint256 deadline
    ) external returns (uint256 amountA, uint256 amountB, uint256 liquidity);
    function addLiquidityNativeToken(
        address token,
        uint256 amountTokenDesired,
        uint256 amountTokenMin,
        uint256 amountNativeTokenMin,
        address to,
        uint256 deadline
    ) external payable returns (uint256 amountToken, uint256 amountNativeToken, uint256 liquidity);
    function removeLiquidity(
        address tokenA,
        address tokenB,
        uint256 liquidity,
        uint256 amountAMin,
        uint256 amountBMin,
        address to,
        uint256 deadline
    ) external returns (uint256 amountA, uint256 amountB);
    function removeLiquidityNativeToken(
        address token,
        uint256 liquidity,
        uint256 amountTokenMin,
        uint256 amountNativeTokenMin,
        address to,
        uint256 deadline
    ) external returns (uint256 amountToken, uint256 amountNativeToken);
    function removeLiquidityWithPermit(
        address tokenA,
        address tokenB,
        uint256 liquidity,
        uint256 amountAMin,
        uint256 amountBMin,
        address to,
        uint256 deadline,
        bool approveMax,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external returns (uint256 amountA, uint256 amountB);
    function removeLiquidityNativeTokenWithPermit(
        address token,
        uint256 liquidity,
        uint256 amountTokenMin,
        uint256 amountNativeTokenMin,
        address to,
        uint256 deadline,
        bool approveMax,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external returns (uint256 amountToken, uint256 amountNativeToken);
```

```
    function swapExactTokensForTokens(
        uint256 amountIn,
        uint256 amountOutMin,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external returns (uint256[] memory amounts);
    function swapTokensForExactTokens(
        uint256 amountOut,
        uint256 amountInMax,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external returns (uint256[] memory amounts);
    function swapExactNativeTokenForTokens(uint256 amountOutMin, address[] calldata path, address to,
        external
        payable
        returns (uint256[] memory amounts);
    function swapTokensForExactNativeToken(
        uint256 amountOut,
        uint256 amountInMax,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external returns (uint256[] memory amounts);
    function swapExactTokensForNativeToken(
        uint256 amountIn,
        uint256 amountOutMin,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external returns (uint256[] memory amounts);
    function swapNativeTokenForExactTokens(uint256 amountOut, address[] calldata path, address to, ui
        external
        payable
        returns (uint256[] memory amounts);

    function quote(uint256 amountA, uint256 reserveA, uint256 reserveB) external pure returns (uint25
    function getAmountOut(uint256 amountIn, uint256 reserveIn, uint256 reserveOut)
        external
        pure
        returns (uint256 amountOut);
    function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut)
        external
        pure
        returns (uint256 amountIn);
    function getAmountsOut(uint256 amountIn, address[] calldata path)
        external
        view
        returns (uint256[] memory amounts);
    function getAmountsIn(uint256 amountOut, address[] calldata path)
        external
        view
        returns (uint256[] memory amounts);
}


// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.17;

// Router interface can be combined into one
// Here we divide it following UniSwap's style for readability

import "./ISwapRouter01.sol";

interface ISwapRouter02 is ISwapRouter01 {
```

```
    function removeLiquidityNativeTokenSupportingFeeOnTransferTokens(
        address token,
        uint256 liquidity,
        uint256 amountTokenMin,
        uint256 amountNativeTokenMin,
        address to,
        uint256 deadline
    ) external returns (uint256 amountNativeToken);
    function removeLiquidityNativeTokenWithPermitSupportingFeeOnTransferTokens(
        address token,
        uint256 liquidity,
        uint256 amountTokenMin,
        uint256 amountNativeTokenMin,
        address to,
        uint256 deadline,
        bool approveMax,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external returns (uint256 amountNativeToken);

    function swapExactTokensForTokensSupportingFeeOnTransferTokens(
        uint256 amountIn,
        uint256 amountOutMin,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external;

    function swapExactNativeTokenForTokensSupportingFeeOnTransferTokens(
        uint256 amountOutMin,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external payable;

    function swapExactTokensForNativeTokenSupportingFeeOnTransferTokens(
        uint256 amountIn,
        uint256 amountOutMin,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external;
}


// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.17;

interface IWNativeToken {
    function deposit() external payable;
    function transfer(address to, uint256 value) external returns (bool);
    function withdraw(uint256) external;
}


// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.17;

// a library for handling binary fixed point numbers (https://en.wikipedia.org/wiki/Q_(number_format)

// range: [0, 2**112 - 1]
// resolution: 1 / 2**112

library UQ112x112 {
    uint224 constant Q112 = 2 ** 112;
```

```
    // encode a uint112 as a UQ112x112
    function encode(uint112 y) internal pure returns (uint224 z) {
        z = uint224(y) * Q112; // never overflows
    }

    // divide a UQ112x112 by a uint112, returning a UQ112x112
    function uqdiv(uint224 x, uint112 y) internal pure returns (uint224 z) {
        z = x / uint224(y);
    }
}


// SPDX-License-Identifier: GPL-3.0-or-later

import {SafeMath} from "../SafeMath.sol";
import {SwapPair} from "../../SwapPair.sol";
import {ISwapPair} from "../../interfaces/ISwapPair.sol";

pragma solidity ^0.8.20;

library SwapRouterLibrary {
    using SafeMath for uint256;

    // returns sorted token addresses, used to handle return values from pairs sorted in this order
    function sortTokens(address tokenA, address tokenB) internal pure returns (address token0, addres
        require(tokenA != tokenB, "SwapLibrary: IDENTICAL_ADDRESSES");
        (token0, token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
        require(token0 != address(0), "SwapLibrary: ZERO_ADDRESS");
    }

    // reading from factory is not the ideal way right now, zksync stack create2 is not fully evm com
    // this method can be tuned later
    function pairFor(address factory, address tokenA, address tokenB) internal view returns (address
        (address token0, address token1) = sortTokens(tokenA, tokenB);
        (bool success, bytes memory data) = factory.staticcall(abi.encodeWithSelector(0xe6a43905, tok
        pair = abi.decode(data, (address));
        require(success, "SwapLibrary: PAIR_FOR_FAILED");
    }

    // fetches and sorts the reserves for a pair
    function getReserves(address factory, address tokenA, address tokenB)
        internal
        view
        returns (uint256 reserveA, uint256 reserveB)
    {
        (address token0,) = sortTokens(tokenA, tokenB);
        (uint256 reserve0, uint256 reserve1,) = ISwapPair(pairFor(factory, tokenA, tokenB)).getReserv
        (reserveA, reserveB) = tokenA == token0 ? (reserve0, reserve1) : (reserve1, reserve0);
    }

    // given some amount of an asset and pair reserves, returns an equivalent amount of the other ass
    // this method is not considering the fee
    function quote(uint256 amountA, uint256 reserveA, uint256 reserveB) internal pure returns (uint25
        require(amountA > 0, "SwapLibrary: INSUFFICIENT_AMOUNT");
        require(reserveA > 0 && reserveB > 0, "SwapLibrary: INSUFFICIENT_LIQUIDITY");
        amountB = amountA.mul(reserveB) / reserveA;
    }

    // given an input amount of an asset and pair reserves, returns the maximum output amount of the
    function getAmountOutWithFee(uint256 amountIn, uint256 reserveIn, uint256 reserveOut, uint256 swa
        internal
        pure
        returns (uint256 amountOut)
    {
        require(amountIn > 0, "SwapLibrary: INSUFFICIENT_INPUT_AMOUNT");
```

```
        require(reserveIn > 0 && reserveOut > 0, "SwapLibrary: INSUFFICIENT_LIQUIDITY");
        uint256 amountInWithFee = amountIn.mul(997 - swapFeeRate);
        uint256 numerator = amountInWithFee.mul(reserveOut);
        uint256 denominator = reserveIn.mul(1000).add(amountInWithFee);
        amountOut = numerator / denominator;
    }

    // given an input amount of an asset and pair reserves, returns the maximum output amount of the
    function getAmountOut(uint256 amountIn, uint256 reserveIn, uint256 reserveOut)
        internal
        pure
        returns (uint256 amountOut)
    {
        amountOut = getAmountOutWithFee(amountIn, reserveIn, reserveOut, 0);
    }

    // given an output amount of an asset and pair reserves, returns a required input amount of the o
    function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut)
        internal
        pure
        returns (uint256 amountIn)
    {
        amountIn = getAmountInWithFee(amountOut, reserveIn, reserveOut, 0);
    }

    function getAmountInWithFee(uint256 amountOut, uint256 reserveIn, uint256 reserveOut, uint256 swa
        internal
        pure
        returns (uint256 amountIn)
    {
        require(amountOut > 0, "SwapLibrary: INSUFFICIENT_OUTPUT_AMOUNT");
        require(reserveIn > 0 && reserveOut > 0, "SwapLibrary: INSUFFICIENT_LIQUIDITY");
        uint256 numerator = reserveIn.mul(amountOut).mul(1000);
        uint256 denominator = reserveOut.sub(amountOut).mul(997 - swapFeeRate);
        amountIn = (numerator / denominator).add(1);
    }

    // performs chained getAmountOut calculations on any number of pairs
    function getAmountsOut(address factory, uint256 amountIn, address[] memory path)
        internal
        view
        returns (uint256[] memory amounts)
    {
        require(path.length >= 2, "SwapLibrary: INVALID_PATH");
        amounts = new uint256[](path.length);
        amounts[0] = amountIn;
        for (uint256 i; i < path.length - 1; i++) {
            address pairAddr = pairFor(factory, path[i], path[i + 1]);
            uint256 swapFeeRate = ISwapPair(pairAddr).swapFeeRate();
            (uint256 reserveIn, uint256 reserveOut) = getReserves(factory, path[i], path[i + 1]);
            amounts[i + 1] = getAmountOutWithFee(amounts[i], reserveIn, reserveOut, swapFeeRate);
        }
    }

    // performs chained getAmountIn calculations on any number of pairs
    function getAmountsIn(address factory, uint256 amountOut, address[] memory path)
        internal
        view
        returns (uint256[] memory amounts)
    {
        require(path.length >= 2, "SwapLibrary: INVALID_PATH");
        amounts = new uint256[](path.length);
        amounts[amounts.length - 1] = amountOut;
        for (uint256 i = path.length - 1; i > 0; i--) {
            address pairAddr = pairFor(factory, path[i - 1], path[i]);
            uint256 swapFeeRate = ISwapPair(pairAddr).swapFeeRate();
```

```
            (uint256 reserveIn, uint256 reserveOut) = getReserves(factory, path[i - 1], path[i]);
            amounts[i - 1] = getAmountInWithFee(amounts[i], reserveIn, reserveOut, swapFeeRate);
        }
    }
}


// SPDX-License-Identifier: GPL-3.0-or-later

pragma solidity ^0.8.20;

// helper methods for interacting with ERC20 tokens and sending ETH that do not consistently return t
library TransferHelper {
    function safeApprove(address token, address to, uint256 value) internal {
        // bytes4(keccak256(bytes('approve(address,uint256)')));
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0x095ea7b3, to, value))
        require(
            success && (data.length == 0 || abi.decode(data, (bool))), "TransferHelper::safeApprove:
        );
    }

    function safeTransfer(address token, address to, uint256 value) internal {
        // bytes4(keccak256(bytes('transfer(address,uint256)')));
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0xa9059cbb, to, value))
        require(
            success && (data.length == 0 || abi.decode(data, (bool))), "TransferHelper::safeTransfer:
        );
    }

    function safeTransferFrom(address token, address from, address to, uint256 value) internal {
        // bytes4(keccak256(bytes('transferFrom(address,address,uint256)')));
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0x23b872dd, from, to, v
        require(
            success && (data.length == 0 || abi.decode(data, (bool))),
            "TransferHelper::transferFrom: transferFrom failed"
        );
    }

    function safeTransferNativeToken(address to, uint256 value) internal {
        (bool success,) = to.call{value: value}(new bytes(0));
        // Keep ETH here
        // TODO: replace with native token
        require(success, "TransferHelper::safeTransferNativeToken: ETH transfer failed");
    }
}


// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.17;

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
import {IWNativeToken} from "../../interfaces/router/IWNativeToken.sol";

contract WNativeToken is ReentrancyGuard, IWNativeToken {
    string private _name;
    string private _symbol;

    uint8 public decimals = 18;

    event Approval(address indexed src, address indexed guy, uint256 wad);
    event Transfer(address indexed src, address indexed dst, uint256 wad);
    event Deposit(address indexed dst, uint256 wad);
    event Withdrawal(address indexed src, uint256 wad);

    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;
```

```
    constructor(string memory name_, string memory symbol_) {
        _name = name_;
        _symbol = symbol_;
    }

    /**
     * @dev Returns the name of the token.
     */
    function name() public view virtual returns (string memory) {
        return _name;
    }

    /**
     * @dev Returns the symbol of the token, usually a shorter version of the
     * name.
     */
    function symbol() public view virtual returns (string memory) {
        return _symbol;
    }

    receive() external payable {
        deposit();
    }

    function deposit() public payable {
        balanceOf[msg.sender] += msg.value;
        emit Deposit(msg.sender, msg.value);
    }

    function withdraw(uint256 wad) public nonReentrant {
        require(balanceOf[msg.sender] >= wad);
        balanceOf[msg.sender] -= wad;
        (bool success, bytes memory data) = payable(msg.sender).call{value: wad}("");
        require(success && (data.length == 0 || abi.decode(data, (bool))), "WNativeToken: WITHDRAW_FA
        emit Withdrawal(msg.sender, wad);
    }

    function totalSupply() public view returns (uint256) {
        return address(this).balance;
    }

    function approve(address guy, uint256 wad) public returns (bool) {
        allowance[msg.sender][guy] = wad;
        emit Approval(msg.sender, guy, wad);
        return true;
    }

    function transfer(address dst, uint256 wad) public returns (bool) {
        return transferFrom(msg.sender, dst, wad);
    }

    function transferFrom(address src, address dst, uint256 wad) public returns (bool) {
        require(balanceOf[src] >= wad);

        if (src != msg.sender && allowance[src][msg.sender] != type(uint256).max) {
            require(allowance[src][msg.sender] >= wad);
            allowance[src][msg.sender] -= wad;
        }

        balanceOf[src] -= wad;
        balanceOf[dst] += wad;

        emit Transfer(src, dst, wad);

        return true;
```

```
    }
}


// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.17;

interface IERC20 {
    event Approval(address indexed owner, address indexed spender, uint256 value);
    event Transfer(address indexed from, address indexed to, uint256 value);

    function name() external view returns (string memory);
    function symbol() external view returns (string memory);
    function decimals() external view returns (uint8);
    function totalSupply() external view returns (uint256);
    function balanceOf(address owner) external view returns (uint256);
    function allowance(address owner, address spender) external view returns (uint256);

    function approve(address spender, uint256 value) external returns (bool);
    function transfer(address to, uint256 value) external returns (bool);
    function transferFrom(address from, address to, uint256 value) external returns (bool);
}


// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.17;

interface ISwapV2Callee {
    function swapV2Call(address sender, uint256 amount0, uint256 amount1, bytes calldata data) extern
}


// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.17;

interface ISwapERC20 {
    event Approval(address indexed owner, address indexed spender, uint256 value);
    event Transfer(address indexed from, address indexed to, uint256 value);

    function name() external pure returns (string memory);
    function symbol() external pure returns (string memory);
    function decimals() external pure returns (uint8);
    function totalSupply() external view returns (uint256);
    function balanceOf(address owner) external view returns (uint256);
    function allowance(address owner, address spender) external view returns (uint256);

    function approve(address spender, uint256 value) external returns (bool);
    function transfer(address to, uint256 value) external returns (bool);
    function transferFrom(address from, address to, uint256 value) external returns (bool);

    function DOMAIN_SEPARATOR() external view returns (bytes32);
    function PERMIT_TYPEHASH() external pure returns (bytes32);
    function nonces(address owner) external view returns (uint256);

    function permit(address owner, address spender, uint256 value, uint256 deadline, uint8 v, bytes32
        external;
}


// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

interface ISwapFactory {
    event PairCreated(address indexed token0, address indexed token1, address pair, uint256);

    function owner() external view returns (address);
```

```
    function feeReceipt() external view returns (address);
    function getPair(address tokenA, address tokenB) external view returns (address pair);
    function allPairs(uint256) external view returns (address pair);
    function allPairsLength() external view returns (uint256);

    function createPair(address tokenA, address tokenB) external returns (address pair);
}


// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity >=0.5.0;

import "./ISwapERC20.sol";
import "./ISwapPairBase.sol";

interface ISwapPair is ISwapERC20, ISwapPairBase {}


// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity >=0.5.0;

import "./IERC20.sol";

interface ISwapPairBase {
    event Mint(address indexed sender, uint256 amount0, uint256 amount1);
    event Burn(address indexed sender, uint256 amount0, uint256 amount1, address indexed to);
    event Swap(
        address indexed sender,
        uint256 amount0In,
        uint256 amount1In,
        uint256 amount0Out,
        uint256 amount1Out,
        address indexed to
    );
    event Sync(uint112 reserve0, uint112 reserve1);

    function MINIMUM_LIQUIDITY() external pure returns (uint256);
    function factory() external view returns (address);
    function token0() external view returns (address);
    function token1() external view returns (address);
    function getReserves() external view returns (uint112 reserve0, uint112 reserve1, uint32 blockTim
    function price0CumulativeLast() external view returns (uint256);
    function price1CumulativeLast() external view returns (uint256);
    function kLast() external view returns (uint256);

    function mint(address to) external returns (uint256 liquidity);
    function burn(address to) external returns (uint256 amount0, uint256 amount1);
    function swap(uint256 amount0Out, uint256 amount1Out, address to, bytes calldata data) external;
    function skim(address to) external;
    function sync() external;

    function initialize(address, address) external;
    function swapFeeRate() external view returns (uint256);
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.0.0) (utils/math/Math.sol)

pragma solidity ^0.8.20;

import {Panic} from "./Panic.sol";
import {SafeCast} from "./SafeCast.sol";

/**
 * @dev Standard math utilities missing in the Solidity language.
```

```
 */
library Math {
    enum Rounding {
        Floor, // Toward negative infinity
        Ceil, // Toward positive infinity
        Trunc, // Toward zero
        Expand // Away from zero

    }

    /**
     * @dev Returns the addition of two unsigned integers, with an success flag (no overflow).
     */
    function tryAdd(uint256 a, uint256 b) internal pure returns (bool success, uint256 result) {
        unchecked {
            uint256 c = a + b;
            if (c < a) return (false, 0);
            return (true, c);
        }
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, with an success flag (no overflow).
     */
    function trySub(uint256 a, uint256 b) internal pure returns (bool success, uint256 result) {
        unchecked {
            if (b > a) return (false, 0);
            return (true, a - b);
        }
    }

    /**
     * @dev Returns the multiplication of two unsigned integers, with an success flag (no overflow).
     */
    function tryMul(uint256 a, uint256 b) internal pure returns (bool success, uint256 result) {
        unchecked {
            // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
            // benefit is lost if 'b' is also tested.
            // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
            if (a == 0) return (true, 0);
            uint256 c = a * b;
            if (c / a != b) return (false, 0);
            return (true, c);
        }
    }

    /**
     * @dev Returns the division of two unsigned integers, with a success flag (no division by zero).
     */
    function tryDiv(uint256 a, uint256 b) internal pure returns (bool success, uint256 result) {
        unchecked {
            if (b == 0) return (false, 0);
            return (true, a / b);
        }
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers, with a success flag (no division
     */
    function tryMod(uint256 a, uint256 b) internal pure returns (bool success, uint256 result) {
        unchecked {
            if (b == 0) return (false, 0);
            return (true, a % b);
        }
    }
```

```
    /**
     * @dev Returns the largest of two numbers.
     */
    function max(uint256 a, uint256 b) internal pure returns (uint256) {
        return a > b ? a : b;
    }

    /**
     * @dev Returns the smallest of two numbers.
     */
    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a < b ? a : b;
    }

    /**
     * @dev Returns the average of two numbers. The result is rounded towards
     * zero.
     */
    function average(uint256 a, uint256 b) internal pure returns (uint256) {
        // (a + b) / 2 can overflow.
        return (a & b) + (a ^ b) / 2;
    }

    /**
     * @dev Returns the ceiling of the division of two numbers.
     *
     * This differs from standard division with `/` in that it rounds towards infinity instead
     * of rounding towards zero.
     */
    function ceilDiv(uint256 a, uint256 b) internal pure returns (uint256) {
        if (b == 0) {
            // Guarantee the same behavior as in a regular Solidity division.
            Panic.panic(Panic.DIVISION_BY_ZERO);
        }

        // The following calculation ensures accurate ceiling division without overflow.
        // Since a is non-zero, (a - 1) / b will not overflow.
        // The largest possible result occurs when (a - 1) / b is type(uint256).max,
        // but the largest value we can obtain is type(uint256).max - 1, which happens
        // when a = type(uint256).max and b = 1.
        unchecked {
            return a == 0 ? 0 : (a - 1) / b + 1;
        }
    }

    /**
     * @dev Calculates floor(x * y / denominator) with full precision. Throws if result overflows a u
     * denominator == 0.
     *
     * Original credit to Remco Bloemen under MIT license (https://xn--2-umb.com/21/muldiv) with furt
     * Uniswap Labs also under MIT license.
     */
    function mulDiv(uint256 x, uint256 y, uint256 denominator) internal pure returns (uint256 result)
        unchecked {
            // 512-bit multiply [prod1 prod0] = x * y. Compute the product mod 2^256 and mod 2^256 - 1, t
            // use the Chinese Remainder Theorem to reconstruct the 512 bit result. The result is sto
            // variables such that product = prod1 * 2^256 + prod0.
            uint256 prod0 = x * y; // Least significant 256 bits of the product
            uint256 prod1; // Most significant 256 bits of the product
            assembly {
                let mm := mulmod(x, y, not(0))
                prod1 := sub(sub(mm, prod0), lt(mm, prod0))
            }

            // Handle non-overflow cases, 256 by 256 division.
            if (prod1 == 0) {
```

```
            // Solidity will revert if denominator == 0, unlike the div opcode on its own.
            // The surrounding unchecked block does not change this fact.
            // See https://docs.soliditylang.org/en/latest/control-structures.html#checked-or-unc
            return prod0 / denominator;
        }

        // Make sure the result is less than 2^256. Also prevents denominator == 0.
        if (denominator <= prod1) {
            Panic.panic(denominator == 0 ? Panic.DIVISION_BY_ZERO : Panic.UNDER_OVERFLOW);
        }

        ///////////////////////////////////////////////
        // 512 by 256 division.
        ///////////////////////////////////////////////

        // Make division exact by subtracting the remainder from [prod1 prod0].
        uint256 remainder;
        assembly {
            // Compute remainder using mulmod.
            remainder := mulmod(x, y, denominator)

            // Subtract 256 bit number from 512 bit number.
            prod1 := sub(prod1, gt(remainder, prod0))
            prod0 := sub(prod0, remainder)
        }

        // Factor powers of two out of denominator and compute largest power of two divisor of de
        // Always >= 1. See https://cs.stackexchange.com/q/138556/92363.

        uint256 twos = denominator & (0 - denominator);
        assembly {
            // Divide denominator by twos.
            denominator := div(denominator, twos)

            // Divide [prod1 prod0] by twos.
            prod0 := div(prod0, twos)

            // Flip twos such that it is 2^256 / twos. If twos is zero, then it becomes one.
            twos := add(div(sub(0, twos), twos), 1)
        }

        // Shift in bits from prod1 into prod0.
        prod0 |= prod1 * twos;

        // Invert denominator mod 2^256. Now that denominator is an odd number, it has an inverse m
        // that denominator * inv ≡ 1 mod 2^256. Compute the inverse by starting with a seed that i
        // four bits. That is, denominator * inv ≡ 1 mod 2^4.
        uint256 inverse = (3 * denominator) ^ 2;

        // Use the Newton-Raphson iteration to improve the precision. Thanks to Hensel's lifting
        // works in modular arithmetic, doubling the correct bits in each step.
        inverse *= 2 - denominator * inverse; // inverse mod 2^8
        inverse *= 2 - denominator * inverse; // inverse mod 2^16
        inverse *= 2 - denominator * inverse; // inverse mod 2^32
        inverse *= 2 - denominator * inverse; // inverse mod 2^64
        inverse *= 2 - denominator * inverse; // inverse mod 2^128
        inverse *= 2 - denominator * inverse; // inverse mod 2^256

        // Because the division is now exact we can divide by multiplying with the modular invers
        // This will give us the correct result modulo 2^256. Since the preconditions guarantee tha
        // less than 2^256, this is the final result. We don't need to compute the high bits of the
        // is no longer required.
        result = prod0 * inverse;
        return result;
    }
}
```

```
    /**
     * @dev Calculates x * y / denominator with full precision, following the selected rounding direc
     */
    function mulDiv(uint256 x, uint256 y, uint256 denominator, Rounding rounding) internal pure retur
        return mulDiv(x, y, denominator) + SafeCast.toUint(unsignedRoundsUp(rounding) && mulmod(x, y,
    }

    /**
     * @dev Calculate the modular multiplicative inverse of a number in Z/nZ.
     *
     * If n is a prime, then Z/nZ is a field. In that case all elements are inversible, expect 0.
     * If n is not a prime, then Z/nZ is not a field, and some elements might not be inversible.
     *
     * If the input value is not inversible, 0 is returned.
     *
     * NOTE: If you know for sure that n is (big) a prime, it may be cheaper to use Ferma's little th
     * inverse using `Math.modExp(a, n - 2, n)`.
     */
    function invMod(uint256 a, uint256 n) internal pure returns (uint256) {
        unchecked {
            if (n == 0) return 0;

            // The inverse modulo is calculated using the Extended Euclidean Algorithm (iterative ver
            // Used to compute integers x and y such that: ax + ny = gcd(a, n).
            // When the gcd is 1, then the inverse of a modulo n exists and it's x.
            // ax + ny = 1
            // ax = 1 + (-y)n
            // ax ≡ 1 (mod n) # x is the inverse of a modulo n

            // If the remainder is 0 the gcd is n right away.
            uint256 remainder = a % n;
            uint256 gcd = n;

            // Therefore the initial coefficients are:
            // ax + ny = gcd(a, n) = n
            // 0a + 1n = n
            int256 x = 0;
            int256 y = 1;

            while (remainder != 0) {
                uint256 quotient = gcd / remainder;

                (gcd, remainder) = (
                    // The old remainder is the next gcd to try.
                    remainder,
                    // Compute the next remainder.
                    // Can't overflow given that (a % gcd) * (gcd // (a % gcd)) <= gcd
                    // where gcd is at most n (capped to type(uint256).max)
                    gcd - remainder * quotient
                );

                (x, y) = (
                    // Increment the coefficient of a.
                    y,
                    // Decrement the coefficient of n.
                    // Can overflow, but the result is casted to uint256 so that the
                    // next value of y is "wrapped around" to a value between 0 and n - 1.
                    x - y * int256(quotient)
                );
            }

            if (gcd != 1) return 0; // No inverse exists.
            return x < 0 ? (n - uint256(-x)) : uint256(x); // Wrap the result if it's negative.
        }
    }
```

```
    /**
     * @dev Returns the modular exponentiation of the specified base, exponent and modulus (b ** e %
     *
     * Requirements:
     * - modulus can't be zero
     * - underlying staticcall to precompile must succeed
     *
     * IMPORTANT: The result is only valid if the underlying call succeeds. When using this function,
     * sure the chain you're using it on supports the precompiled contract for modular exponentiation
     * at address 0x05 as specified in https://eips.ethereum.org/EIPS/eip-198[EIP-198]. Otherwise,
     * the underlying function will succeed given the lack of a revert, but the result may be incorre
     * interpreted as 0.
     */
    function modExp(uint256 b, uint256 e, uint256 m) internal view returns (uint256) {
        (bool success, uint256 result) = tryModExp(b, e, m);
        if (!success) {
            Panic.panic(Panic.DIVISION_BY_ZERO);
        }
        return result;
    }

    /**
     * @dev Returns the modular exponentiation of the specified base, exponent and modulus (b ** e %
     * It includes a success flag indicating if the operation succeeded. Operation will be marked has
     * to operate modulo 0 or if the underlying precompile reverted.
     *
     * IMPORTANT: The result is only valid if the success flag is true. When using this function, mak
     * you're using it on supports the precompiled contract for modular exponentiation at address 0x0
     * https://eips.ethereum.org/EIPS/eip-198[EIP-198]. Otherwise, the underlying function will succe
     * of a revert, but the result may be incorrectly interpreted as 0.
     */
    function tryModExp(uint256 b, uint256 e, uint256 m) internal view returns (bool success, uint256
        if (m == 0) return (false, 0);
        /// @solidity memory-safe-assembly
        assembly {
            let ptr := mload(0x40)
            // | Offset    | Content    | Content (Hex)
            // |-----------|------------|--------------------------------------------------------------
            // | 0x00:0x1f | size of b  | 0x0000000000000000000000000000000000000000000000000000000000000000
            // | 0x20:0x3f | size of e  | 0x0000000000000000000000000000000000000000000000000000000000000000
            // | 0x40:0x5f | size of m  | 0x0000000000000000000000000000000000000000000000000000000000000000
            // | 0x60:0x7f | value of b | 0x<.................................................
            // | 0x80:0x9f | value of e | 0x<.................................................
            // | 0xa0:0xbf | value of m | 0x<.................................................
            mstore(ptr, 0x20)
            mstore(add(ptr, 0x20), 0x20)
            mstore(add(ptr, 0x40), 0x20)
            mstore(add(ptr, 0x60), b)
            mstore(add(ptr, 0x80), e)
            mstore(add(ptr, 0xa0), m)

            // Given the result < m, it's guaranteed to fit in 32 bytes,
            // so we can use the memory scratch space located at offset 0.
            success := staticcall(gas(), 0x05, ptr, 0xc0, 0x00, 0x20)
            result := mload(0x00)
        }
    }

    /**
     * @dev Variant of {modExp} that supports inputs of arbitrary length.
     */
    function modExp(bytes memory b, bytes memory e, bytes memory m) internal view returns (bytes memo
        (bool success, bytes memory result) = tryModExp(b, e, m);
        if (!success) {
            Panic.panic(Panic.DIVISION_BY_ZERO);
```

```
        }
        return result;
    }

    /**
     * @dev Variant of {tryModExp} that supports inputs of arbitrary length.
     */
    function tryModExp(bytes memory b, bytes memory e, bytes memory m)
        internal
        view
        returns (bool success, bytes memory result)
    {
        if (_zeroBytes(m)) return (false, new bytes(0));

        uint256 mLen = m.length;

        // Encode call args in result and move the free memory pointer
        result = abi.encodePacked(b.length, e.length, mLen, b, e, m);

        /// @solidity memory-safe-assembly
        assembly {
            let dataPtr := add(result, 0x20)
            // Write result on top of args to avoid allocating extra memory.
            success := staticcall(gas(), 0x05, dataPtr, mload(result), dataPtr, mLen)
            // Overwrite the length.
            // result.length > returndatasize() is guaranteed because returndatasize() == m.length
            mstore(result, mLen)
            // Set the memory pointer after the returned data.
            mstore(0x40, add(dataPtr, mLen))
        }
    }

    /**
     * @dev Returns whether the provided byte array is zero.
     */
    function _zeroBytes(bytes memory byteArray) private pure returns (bool) {
        for (uint256 i = 0; i < byteArray.length; ++i) {
            if (byteArray[i] != 0) {
                return false;
            }
        }
        return true;
    }

    /**
     * @dev Returns the square root of a number. If the number is not a perfect square, the value is
     * towards zero.
     *
     * This method is based on Newton's method for computing square roots; the algorithm is restricte
     * using integer operations.
     */
    function sqrt(uint256 a) internal pure returns (uint256) {
        unchecked {
            // Take care of easy edge cases when a == 0 or a == 1
            if (a <= 1) {
                return a;
            }

            // In this function, we use Newton's method to get a root of `f(x) := x² - a`. It involve
            // sequence x_n that converges toward sqrt(a). For each iteration x_n, we also define the
            // the current value as `ε_n = | x_n - sqrt(a) |`.
            //
            // For our first estimation, we consider `e` the smallest power of 2 which is bigger than
            // of the target. (i.e. `2**(e-1) ≤ sqrt(a) < 2**e`). We know that `e ≤ 128` because `(2¹
            // bigger than any uint256.
            //
```

```
        // By noticing that
        // `2**(e-1) ≤ sqrt(a) < 2**e → (2**(e-1))² ≤ a < (2**e)² → 2**(2*e-2) ≤ a < 2**(2*e)`
        // we can deduce that `e - 1` is `log2(a) / 2`. We can thus compute `x_n = 2**(e-1)` usin
        // to the msb function.
        uint256 aa = a;
        uint256 xn = 1;

        if (aa >= (1 << 128)) {
            aa >>= 128;
            xn <<= 64;
        }
        if (aa >= (1 << 64)) {
            aa >>= 64;
            xn <<= 32;
        }
        if (aa >= (1 << 32)) {
            aa >>= 32;
            xn <<= 16;
        }
        if (aa >= (1 << 16)) {
            aa >>= 16;
            xn <<= 8;
        }
        if (aa >= (1 << 8)) {
            aa >>= 8;
            xn <<= 4;
        }
        if (aa >= (1 << 4)) {
            aa >>= 4;
            xn <<= 2;
        }
        if (aa >= (1 << 2)) {
            xn <<= 1;
        }

        // We now have x_n such that `x_n = 2**(e-1) ≤ sqrt(a) < 2**e = 2 * x_n`. This implies ε_
        //
        // We can refine our estimation by noticing that the the middle of that interval minimize
        // If we move x_n to equal 2**(e-1) + 2**(e-2), then we reduce the error to ε_n ≤ 2**(e-2
        // This is going to be our x_0 (and ε_0)
        xn = (3 * xn) >> 1; // ε_0 := | x_0 - sqrt(a) | ≤ 2**(e-2)

        // From here, Newton's method give us:
        // x_{n+1} = (x_n + a / x_n) / 2
        //
        // One should note that:
        // x_{n+1}² - a = ((x_n + a / x_n) / 2)² - a
        //              = ((x_n² + a) / (2 * x_n))² - a
        //              = (x_n⁴ + 2 * a * x_n² + a²) / (4 * x_n²) - a
        //              = (x_n⁴ + 2 * a * x_n² + a² - 4 * a * x_n²) / (4 * x_n²)
        //              = (x_n⁴ - 2 * a * x_n² + a²) / (4 * x_n²)
        //              = (x_n² - a)² / (2 * x_n)²
        //              = ((x_n² - a) / (2 * x_n))²
        //              ≥ 0
        // Which proves that for all n ≥ 1, sqrt(a) ≤ x_n
        //
        // This gives us the proof of quadratic convergence of the sequence:
        // ε_{n+1} = | x_{n+1} - sqrt(a) |
        //         = | (x_n + a / x_n) / 2 - sqrt(a) |
        //         = | (x_n² + a - 2*x_n*sqrt(a)) / (2 * x_n) |
        //         = | (x_n - sqrt(a))² / (2 * x_n) |
        //         = | ε_n² / (2 * x_n) |
        //         = ε_n² / | (2 * x_n) |
        //
        // For the first iteration, we have a special case where x_0 is known:
        // ε_1 = ε_0² / | (2 * x_0) |
```

```
            //     ≤ (2**(e-2))² / (2 * (2**(e-1) + 2**(e-2)))
            //     ≤ 2**(2*e-4) / (3 * 2**(e-1))
            //     ≤ 2**(e-3) / 3
            //     ≤ 2**(e-3-log2(3))
            //     ≤ 2**(e-4.5)
            //
            // For the following iterations, we use the fact that, 2**(e-1) ≤ sqrt(a) ≤ x_n:
            // ε_{n+1} = ε_n² / | (2 * x_n) |
            //         ≤ (2**(e-k))² / (2 * 2**(e-1))
            //         ≤ 2**(2*e-2*k) / 2**e
            //         ≤ 2**(e-2*k)
            xn = (xn + a / xn) >> 1; // ε_1 := | x_1 - sqrt(a) | ≤ 2**(e-4.5)  -- special case, see a
            xn = (xn + a / xn) >> 1; // ε_2 := | x_2 - sqrt(a) | ≤ 2**(e-9)    -- general case with k
            xn = (xn + a / xn) >> 1; // ε_3 := | x_3 - sqrt(a) | ≤ 2**(e-18)   -- general case with k
            xn = (xn + a / xn) >> 1; // ε_4 := | x_4 - sqrt(a) | ≤ 2**(e-36)   -- general case with k
            xn = (xn + a / xn) >> 1; // ε_5 := | x_5 - sqrt(a) | ≤ 2**(e-72)   -- general case with k
            xn = (xn + a / xn) >> 1; // ε_6 := | x_6 - sqrt(a) | ≤ 2**(e-144)  -- general case with k

            // Because e ≤ 128 (as discussed during the first estimation phase), we know have reached
            // ε_6 ≤ 2**(e-144) < 1. Given we're operating on integers, then we can ensure that xn is
            // sqrt(a) or sqrt(a) + 1.
            return xn - SafeCast.toUint(xn > a / xn);
        }
    }

    /**
     * @dev Calculates sqrt(a), following the selected rounding direction.
     */
    function sqrt(uint256 a, Rounding rounding) internal pure returns (uint256) {
        unchecked {
            uint256 result = sqrt(a);
            return result + SafeCast.toUint(unsignedRoundsUp(rounding) && result * result < a);
        }
    }

    /**
     * @dev Return the log in base 2 of a positive value rounded towards zero.
     * Returns 0 if given 0.
     */
    function log2(uint256 value) internal pure returns (uint256) {
        uint256 result = 0;
        uint256 exp;
        unchecked {
            exp = 128 * SafeCast.toUint(value > (1 << 128) - 1);
            value >>= exp;
            result += exp;

            exp = 64 * SafeCast.toUint(value > (1 << 64) - 1);
            value >>= exp;
            result += exp;

            exp = 32 * SafeCast.toUint(value > (1 << 32) - 1);
            value >>= exp;
            result += exp;

            exp = 16 * SafeCast.toUint(value > (1 << 16) - 1);
            value >>= exp;
            result += exp;

            exp = 8 * SafeCast.toUint(value > (1 << 8) - 1);
            value >>= exp;
            result += exp;

            exp = 4 * SafeCast.toUint(value > (1 << 4) - 1);
            value >>= exp;
            result += exp;
```

```
            exp = 2 * SafeCast.toUint(value > (1 << 2) - 1);
            value >>= exp;
            result += exp;

            result += SafeCast.toUint(value > 1);
        }
        return result;
    }

    /**
     * @dev Return the log in base 2, following the selected rounding direction, of a positive value.
     * Returns 0 if given 0.
     */
    function log2(uint256 value, Rounding rounding) internal pure returns (uint256) {
        unchecked {
            uint256 result = log2(value);
            return result + SafeCast.toUint(unsignedRoundsUp(rounding) && 1 << result < value);
        }
    }

    /**
     * @dev Return the log in base 10 of a positive value rounded towards zero.
     * Returns 0 if given 0.
     */
    function log10(uint256 value) internal pure returns (uint256) {
        uint256 result = 0;
        unchecked {
            if (value >= 10 ** 64) {
                value /= 10 ** 64;
                result += 64;
            }
            if (value >= 10 ** 32) {
                value /= 10 ** 32;
                result += 32;
            }
            if (value >= 10 ** 16) {
                value /= 10 ** 16;
                result += 16;
            }
            if (value >= 10 ** 8) {
                value /= 10 ** 8;
                result += 8;
            }
            if (value >= 10 ** 4) {
                value /= 10 ** 4;
                result += 4;
            }
            if (value >= 10 ** 2) {
                value /= 10 ** 2;
                result += 2;
            }
            if (value >= 10 ** 1) {
                result += 1;
            }
        }
        return result;
    }

    /**
     * @dev Return the log in base 10, following the selected rounding direction, of a positive value
     * Returns 0 if given 0.
     */
    function log10(uint256 value, Rounding rounding) internal pure returns (uint256) {
        unchecked {
            uint256 result = log10(value);
```

```
            return result + SafeCast.toUint(unsignedRoundsUp(rounding) && 10 ** result < value);
        }
    }

    /**
     * @dev Return the log in base 256 of a positive value rounded towards zero.
     * Returns 0 if given 0.
     *
     * Adding one to the result gives the number of pairs of hex symbols needed to represent `value`
     */
    function log256(uint256 value) internal pure returns (uint256) {
        uint256 result = 0;
        uint256 isGt;
        unchecked {
            isGt = SafeCast.toUint(value > (1 << 128) - 1);
            value >>= isGt * 128;
            result += isGt * 16;

            isGt = SafeCast.toUint(value > (1 << 64) - 1);
            value >>= isGt * 64;
            result += isGt * 8;

            isGt = SafeCast.toUint(value > (1 << 32) - 1);
            value >>= isGt * 32;
            result += isGt * 4;

            isGt = SafeCast.toUint(value > (1 << 16) - 1);
            value >>= isGt * 16;
            result += isGt * 2;

            result += SafeCast.toUint(value > (1 << 8) - 1);
        }
        return result;
    }

    /**
     * @dev Return the log in base 256, following the selected rounding direction, of a positive valu
     * Returns 0 if given 0.
     */
    function log256(uint256 value, Rounding rounding) internal pure returns (uint256) {
        unchecked {
            uint256 result = log256(value);
            return result + SafeCast.toUint(unsignedRoundsUp(rounding) && 1 << (result << 3) < value)
        }
    }

    /**
     * @dev Returns whether a provided rounding mode is considered rounding up for unsigned integers.
     */
    function unsignedRoundsUp(Rounding rounding) internal pure returns (bool) {
        return uint8(rounding) % 2 == 1;
    }
}


// SPDX-License-Identifier: MIT

pragma solidity ^0.8.17;

/**
 * @dev Helper library for emitting standardized panic codes.
 *
 * solidity
 * contract Example {
 *      using Panic for uint256;
 *
```

```
 *      // Use any of the declared internal constants
 *      function foo() { Panic.GENERIC.panic(); }
 *
 *      // Alternatively
 *      function foo() { Panic.panic(Panic.GENERIC); }
 * }
 *
 *
 * Follows the list from https://github.com/ethereum/solidity/blob/v0.8.24/libsolutil/ErrorCodes.h[li
 */
// slither-disable-next-line unused-state
library Panic {
    /// @dev generic / unspecified error
    uint256 internal constant GENERIC = 0x00;
    /// @dev used by the assert() builtin
    uint256 internal constant ASSERT = 0x01;
    /// @dev arithmetic underflow or overflow
    uint256 internal constant UNDER_OVERFLOW = 0x11;
    /// @dev division or modulo by zero
    uint256 internal constant DIVISION_BY_ZERO = 0x12;
    /// @dev enum conversion error
    uint256 internal constant ENUM_CONVERSION_ERROR = 0x21;
    /// @dev invalid encoding in storage
    uint256 internal constant STORAGE_ENCODING_ERROR = 0x22;
    /// @dev empty array pop
    uint256 internal constant EMPTY_ARRAY_POP = 0x31;
    /// @dev array out of bounds access
    uint256 internal constant ARRAY_OUT_OF_BOUNDS = 0x32;
    /// @dev resource error (too large allocation or too large array)
    uint256 internal constant RESOURCE_ERROR = 0x41;
    /// @dev calling invalid internal function
    uint256 internal constant INVALID_INTERNAL_FUNCTION = 0x51;

    /// @dev Reverts with a panic code. Recommended to use with
    /// the internal constants with predefined codes.
    function panic(uint256 code) internal pure {
        /// @solidity memory-safe-assembly
        assembly {
            mstore(0x00, 0x4e487b71)
            mstore(0x20, code)
            revert(0x1c, 0x24)
        }
    }
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.0.0) (utils/math/SafeCast.sol)
// This file was procedurally generated from scripts/generate/templates/SafeCast.js.

pragma solidity ^0.8.20;

/**
 * @dev Wrappers over Solidity's uintXX/intXX/bool casting operators with added overflow
 * checks.
 *
 * Downcasting from uint256/int256 in Solidity does not revert on overflow. This can
 * easily result in undesired exploitation or bugs, since developers usually
 * assume that overflows raise errors. `SafeCast` restores this intuition by
 * reverting the transaction when such an operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeCast {
    /**
```

```
     * @dev Value doesn't fit in an uint of `bits` size.
     */
    error SafeCastOverflowedUintDowncast(uint8 bits, uint256 value);

    /**
     * @dev An int value doesn't fit in an uint of `bits` size.
     */
    error SafeCastOverflowedIntToUint(int256 value);

    /**
     * @dev Value doesn't fit in an int of `bits` size.
     */
    error SafeCastOverflowedIntDowncast(uint8 bits, int256 value);

    /**
     * @dev An uint value doesn't fit in an int of `bits` size.
     */
    error SafeCastOverflowedUintToInt(uint256 value);

    /**
     * @dev Returns the downcasted uint248 from uint256, reverting on
     * overflow (when the input is greater than largest uint248).
     *
     * Counterpart to Solidity's `uint248` operator.
     *
     * Requirements:
     *
     * - input must fit into 248 bits
     */
    function toUint248(uint256 value) internal pure returns (uint248) {
        if (value > type(uint248).max) {
            revert SafeCastOverflowedUintDowncast(248, value);
        }
        return uint248(value);
    }

    /**
     * @dev Returns the downcasted uint240 from uint256, reverting on
     * overflow (when the input is greater than largest uint240).
     *
     * Counterpart to Solidity's `uint240` operator.
     *
     * Requirements:
     *
     * - input must fit into 240 bits
     */
    function toUint240(uint256 value) internal pure returns (uint240) {
        if (value > type(uint240).max) {
            revert SafeCastOverflowedUintDowncast(240, value);
        }
        return uint240(value);
    }

    /**
     * @dev Returns the downcasted uint232 from uint256, reverting on
     * overflow (when the input is greater than largest uint232).
     *
     * Counterpart to Solidity's `uint232` operator.
     *
     * Requirements:
     *
     * - input must fit into 232 bits
     */
    function toUint232(uint256 value) internal pure returns (uint232) {
        if (value > type(uint232).max) {
            revert SafeCastOverflowedUintDowncast(232, value);
```

```
        }
        return uint232(value);
    }

    /**
     * @dev Returns the downcasted uint224 from uint256, reverting on
     * overflow (when the input is greater than largest uint224).
     *
     * Counterpart to Solidity's `uint224` operator.
     *
     * Requirements:
     *
     * - input must fit into 224 bits
     */
    function toUint224(uint256 value) internal pure returns (uint224) {
        if (value > type(uint224).max) {
            revert SafeCastOverflowedUintDowncast(224, value);
        }
        return uint224(value);
    }

    /**
     * @dev Returns the downcasted uint216 from uint256, reverting on
     * overflow (when the input is greater than largest uint216).
     *
     * Counterpart to Solidity's `uint216` operator.
     *
     * Requirements:
     *
     * - input must fit into 216 bits
     */
    function toUint216(uint256 value) internal pure returns (uint216) {
        if (value > type(uint216).max) {
            revert SafeCastOverflowedUintDowncast(216, value);
        }
        return uint216(value);
    }

    /**
     * @dev Returns the downcasted uint208 from uint256, reverting on
     * overflow (when the input is greater than largest uint208).
     *
     * Counterpart to Solidity's `uint208` operator.
     *
     * Requirements:
     *
     * - input must fit into 208 bits
     */
    function toUint208(uint256 value) internal pure returns (uint208) {
        if (value > type(uint208).max) {
            revert SafeCastOverflowedUintDowncast(208, value);
        }
        return uint208(value);
    }

    /**
     * @dev Returns the downcasted uint200 from uint256, reverting on
     * overflow (when the input is greater than largest uint200).
     *
     * Counterpart to Solidity's `uint200` operator.
     *
     * Requirements:
     *
     * - input must fit into 200 bits
     */
    function toUint200(uint256 value) internal pure returns (uint200) {
```

```
        if (value > type(uint200).max) {
            revert SafeCastOverflowedUintDowncast(200, value);
        }
        return uint200(value);
    }

    /**
     * @dev Returns the downcasted uint192 from uint256, reverting on
     * overflow (when the input is greater than largest uint192).
     *
     * Counterpart to Solidity's `uint192` operator.
     *
     * Requirements:
     *
     * - input must fit into 192 bits
     */
    function toUint192(uint256 value) internal pure returns (uint192) {
        if (value > type(uint192).max) {
            revert SafeCastOverflowedUintDowncast(192, value);
        }
        return uint192(value);
    }

    /**
     * @dev Returns the downcasted uint184 from uint256, reverting on
     * overflow (when the input is greater than largest uint184).
     *
     * Counterpart to Solidity's `uint184` operator.
     *
     * Requirements:
     *
     * - input must fit into 184 bits
     */
    function toUint184(uint256 value) internal pure returns (uint184) {
        if (value > type(uint184).max) {
            revert SafeCastOverflowedUintDowncast(184, value);
        }
        return uint184(value);
    }

    /**
     * @dev Returns the downcasted uint176 from uint256, reverting on
     * overflow (when the input is greater than largest uint176).
     *
     * Counterpart to Solidity's `uint176` operator.
     *
     * Requirements:
     *
     * - input must fit into 176 bits
     */
    function toUint176(uint256 value) internal pure returns (uint176) {
        if (value > type(uint176).max) {
            revert SafeCastOverflowedUintDowncast(176, value);
        }
        return uint176(value);
    }

    /**
     * @dev Returns the downcasted uint168 from uint256, reverting on
     * overflow (when the input is greater than largest uint168).
     *
     * Counterpart to Solidity's `uint168` operator.
     *
     * Requirements:
     *
     * - input must fit into 168 bits
```

```
     */
    function toUint168(uint256 value) internal pure returns (uint168) {
        if (value > type(uint168).max) {
            revert SafeCastOverflowedUintDowncast(168, value);
        }
        return uint168(value);
    }

    /**
     * @dev Returns the downcasted uint160 from uint256, reverting on
     * overflow (when the input is greater than largest uint160).
     *
     * Counterpart to Solidity's `uint160` operator.
     *
     * Requirements:
     *
     * - input must fit into 160 bits
     */
    function toUint160(uint256 value) internal pure returns (uint160) {
        if (value > type(uint160).max) {
            revert SafeCastOverflowedUintDowncast(160, value);
        }
        return uint160(value);
    }

    /**
     * @dev Returns the downcasted uint152 from uint256, reverting on
     * overflow (when the input is greater than largest uint152).
     *
     * Counterpart to Solidity's `uint152` operator.
     *
     * Requirements:
     *
     * - input must fit into 152 bits
     */
    function toUint152(uint256 value) internal pure returns (uint152) {
        if (value > type(uint152).max) {
            revert SafeCastOverflowedUintDowncast(152, value);
        }
        return uint152(value);
    }

    /**
     * @dev Returns the downcasted uint144 from uint256, reverting on
     * overflow (when the input is greater than largest uint144).
     *
     * Counterpart to Solidity's `uint144` operator.
     *
     * Requirements:
     *
     * - input must fit into 144 bits
     */
    function toUint144(uint256 value) internal pure returns (uint144) {
        if (value > type(uint144).max) {
            revert SafeCastOverflowedUintDowncast(144, value);
        }
        return uint144(value);
    }

    /**
     * @dev Returns the downcasted uint136 from uint256, reverting on
     * overflow (when the input is greater than largest uint136).
     *
     * Counterpart to Solidity's `uint136` operator.
     *
     * Requirements:
```

```
     *
     * - input must fit into 136 bits
     */
    function toUint136(uint256 value) internal pure returns (uint136) {
        if (value > type(uint136).max) {
            revert SafeCastOverflowedUintDowncast(136, value);
        }
        return uint136(value);
    }

    /**
     * @dev Returns the downcasted uint128 from uint256, reverting on
     * overflow (when the input is greater than largest uint128).
     *
     * Counterpart to Solidity's `uint128` operator.
     *
     * Requirements:
     *
     * - input must fit into 128 bits
     */
    function toUint128(uint256 value) internal pure returns (uint128) {
        if (value > type(uint128).max) {
            revert SafeCastOverflowedUintDowncast(128, value);
        }
        return uint128(value);
    }

    /**
     * @dev Returns the downcasted uint120 from uint256, reverting on
     * overflow (when the input is greater than largest uint120).
     *
     * Counterpart to Solidity's `uint120` operator.
     *
     * Requirements:
     *
     * - input must fit into 120 bits
     */
    function toUint120(uint256 value) internal pure returns (uint120) {
        if (value > type(uint120).max) {
            revert SafeCastOverflowedUintDowncast(120, value);
        }
        return uint120(value);
    }

    /**
     * @dev Returns the downcasted uint112 from uint256, reverting on
     * overflow (when the input is greater than largest uint112).
     *
     * Counterpart to Solidity's `uint112` operator.
     *
     * Requirements:
     *
     * - input must fit into 112 bits
     */
    function toUint112(uint256 value) internal pure returns (uint112) {
        if (value > type(uint112).max) {
            revert SafeCastOverflowedUintDowncast(112, value);
        }
        return uint112(value);
    }

    /**
     * @dev Returns the downcasted uint104 from uint256, reverting on
     * overflow (when the input is greater than largest uint104).
     *
     * Counterpart to Solidity's `uint104` operator.
```

```
     *
     * Requirements:
     *
     * - input must fit into 104 bits
     */
    function toUint104(uint256 value) internal pure returns (uint104) {
        if (value > type(uint104).max) {
            revert SafeCastOverflowedUintDowncast(104, value);
        }
        return uint104(value);
    }

    /**
     * @dev Returns the downcasted uint96 from uint256, reverting on
     * overflow (when the input is greater than largest uint96).
     *
     * Counterpart to Solidity's `uint96` operator.
     *
     * Requirements:
     *
     * - input must fit into 96 bits
     */
    function toUint96(uint256 value) internal pure returns (uint96) {
        if (value > type(uint96).max) {
            revert SafeCastOverflowedUintDowncast(96, value);
        }
        return uint96(value);
    }

    /**
     * @dev Returns the downcasted uint88 from uint256, reverting on
     * overflow (when the input is greater than largest uint88).
     *
     * Counterpart to Solidity's `uint88` operator.
     *
     * Requirements:
     *
     * - input must fit into 88 bits
     */
    function toUint88(uint256 value) internal pure returns (uint88) {
        if (value > type(uint88).max) {
            revert SafeCastOverflowedUintDowncast(88, value);
        }
        return uint88(value);
    }

    /**
     * @dev Returns the downcasted uint80 from uint256, reverting on
     * overflow (when the input is greater than largest uint80).
     *
     * Counterpart to Solidity's `uint80` operator.
     *
     * Requirements:
     *
     * - input must fit into 80 bits
     */
    function toUint80(uint256 value) internal pure returns (uint80) {
        if (value > type(uint80).max) {
            revert SafeCastOverflowedUintDowncast(80, value);
        }
        return uint80(value);
    }

    /**
     * @dev Returns the downcasted uint72 from uint256, reverting on
     * overflow (when the input is greater than largest uint72).
```

```
     *
     * Counterpart to Solidity's `uint72` operator.
     *
     * Requirements:
     *
     * - input must fit into 72 bits
     */
    function toUint72(uint256 value) internal pure returns (uint72) {
        if (value > type(uint72).max) {
            revert SafeCastOverflowedUintDowncast(72, value);
        }
        return uint72(value);
    }

    /**
     * @dev Returns the downcasted uint64 from uint256, reverting on
     * overflow (when the input is greater than largest uint64).
     *
     * Counterpart to Solidity's `uint64` operator.
     *
     * Requirements:
     *
     * - input must fit into 64 bits
     */
    function toUint64(uint256 value) internal pure returns (uint64) {
        if (value > type(uint64).max) {
            revert SafeCastOverflowedUintDowncast(64, value);
        }
        return uint64(value);
    }

    /**
     * @dev Returns the downcasted uint56 from uint256, reverting on
     * overflow (when the input is greater than largest uint56).
     *
     * Counterpart to Solidity's `uint56` operator.
     *
     * Requirements:
     *
     * - input must fit into 56 bits
     */
    function toUint56(uint256 value) internal pure returns (uint56) {
        if (value > type(uint56).max) {
            revert SafeCastOverflowedUintDowncast(56, value);
        }
        return uint56(value);
    }

    /**
     * @dev Returns the downcasted uint48 from uint256, reverting on
     * overflow (when the input is greater than largest uint48).
     *
     * Counterpart to Solidity's `uint48` operator.
     *
     * Requirements:
     *
     * - input must fit into 48 bits
     */
    function toUint48(uint256 value) internal pure returns (uint48) {
        if (value > type(uint48).max) {
            revert SafeCastOverflowedUintDowncast(48, value);
        }
        return uint48(value);
    }

    /**
```

```
     * @dev Returns the downcasted uint40 from uint256, reverting on
     * overflow (when the input is greater than largest uint40).
     *
     * Counterpart to Solidity's `uint40` operator.
     *
     * Requirements:
     *
     * - input must fit into 40 bits
     */
    function toUint40(uint256 value) internal pure returns (uint40) {
        if (value > type(uint40).max) {
            revert SafeCastOverflowedUintDowncast(40, value);
        }
        return uint40(value);
    }

    /**
     * @dev Returns the downcasted uint32 from uint256, reverting on
     * overflow (when the input is greater than largest uint32).
     *
     * Counterpart to Solidity's `uint32` operator.
     *
     * Requirements:
     *
     * - input must fit into 32 bits
     */
    function toUint32(uint256 value) internal pure returns (uint32) {
        if (value > type(uint32).max) {
            revert SafeCastOverflowedUintDowncast(32, value);
        }
        return uint32(value);
    }

    /**
     * @dev Returns the downcasted uint24 from uint256, reverting on
     * overflow (when the input is greater than largest uint24).
     *
     * Counterpart to Solidity's `uint24` operator.
     *
     * Requirements:
     *
     * - input must fit into 24 bits
     */
    function toUint24(uint256 value) internal pure returns (uint24) {
        if (value > type(uint24).max) {
            revert SafeCastOverflowedUintDowncast(24, value);
        }
        return uint24(value);
    }

    /**
     * @dev Returns the downcasted uint16 from uint256, reverting on
     * overflow (when the input is greater than largest uint16).
     *
     * Counterpart to Solidity's `uint16` operator.
     *
     * Requirements:
     *
     * - input must fit into 16 bits
     */
    function toUint16(uint256 value) internal pure returns (uint16) {
        if (value > type(uint16).max) {
            revert SafeCastOverflowedUintDowncast(16, value);
        }
        return uint16(value);
    }
```

```
    /**
     * @dev Returns the downcasted uint8 from uint256, reverting on
     * overflow (when the input is greater than largest uint8).
     *
     * Counterpart to Solidity's `uint8` operator.
     *
     * Requirements:
     *
     * - input must fit into 8 bits
     */
    function toUint8(uint256 value) internal pure returns (uint8) {
        if (value > type(uint8).max) {
            revert SafeCastOverflowedUintDowncast(8, value);
        }
        return uint8(value);
    }

    /**
     * @dev Converts a signed int256 into an unsigned uint256.
     *
     * Requirements:
     *
     * - input must be greater than or equal to 0.
     */
    function toUint256(int256 value) internal pure returns (uint256) {
        if (value < 0) {
            revert SafeCastOverflowedIntToUint(value);
        }
        return uint256(value);
    }

    /**
     * @dev Returns the downcasted int248 from int256, reverting on
     * overflow (when the input is less than smallest int248 or
     * greater than largest int248).
     *
     * Counterpart to Solidity's `int248` operator.
     *
     * Requirements:
     *
     * - input must fit into 248 bits
     */
    function toInt248(int256 value) internal pure returns (int248 downcasted) {
        downcasted = int248(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(248, value);
        }
    }

    /**
     * @dev Returns the downcasted int240 from int256, reverting on
     * overflow (when the input is less than smallest int240 or
     * greater than largest int240).
     *
     * Counterpart to Solidity's `int240` operator.
     *
     * Requirements:
     *
     * - input must fit into 240 bits
     */
    function toInt240(int256 value) internal pure returns (int240 downcasted) {
        downcasted = int240(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(240, value);
        }
```

```
    }

    /**
     * @dev Returns the downcasted int232 from int256, reverting on
     * overflow (when the input is less than smallest int232 or
     * greater than largest int232).
     *
     * Counterpart to Solidity's `int232` operator.
     *
     * Requirements:
     *
     * - input must fit into 232 bits
     */
    function toInt232(int256 value) internal pure returns (int232 downcasted) {
        downcasted = int232(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(232, value);
        }
    }

    /**
     * @dev Returns the downcasted int224 from int256, reverting on
     * overflow (when the input is less than smallest int224 or
     * greater than largest int224).
     *
     * Counterpart to Solidity's `int224` operator.
     *
     * Requirements:
     *
     * - input must fit into 224 bits
     */
    function toInt224(int256 value) internal pure returns (int224 downcasted) {
        downcasted = int224(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(224, value);
        }
    }

    /**
     * @dev Returns the downcasted int216 from int256, reverting on
     * overflow (when the input is less than smallest int216 or
     * greater than largest int216).
     *
     * Counterpart to Solidity's `int216` operator.
     *
     * Requirements:
     *
     * - input must fit into 216 bits
     */
    function toInt216(int256 value) internal pure returns (int216 downcasted) {
        downcasted = int216(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(216, value);
        }
    }

    /**
     * @dev Returns the downcasted int208 from int256, reverting on
     * overflow (when the input is less than smallest int208 or
     * greater than largest int208).
     *
     * Counterpart to Solidity's `int208` operator.
     *
     * Requirements:
     *
     * - input must fit into 208 bits
```

```
     */
    function toInt208(int256 value) internal pure returns (int208 downcasted) {
        downcasted = int208(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(208, value);
        }
    }

    /**
     * @dev Returns the downcasted int200 from int256, reverting on
     * overflow (when the input is less than smallest int200 or
     * greater than largest int200).
     *
     * Counterpart to Solidity's `int200` operator.
     *
     * Requirements:
     *
     * - input must fit into 200 bits
     */
    function toInt200(int256 value) internal pure returns (int200 downcasted) {
        downcasted = int200(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(200, value);
        }
    }

    /**
     * @dev Returns the downcasted int192 from int256, reverting on
     * overflow (when the input is less than smallest int192 or
     * greater than largest int192).
     *
     * Counterpart to Solidity's `int192` operator.
     *
     * Requirements:
     *
     * - input must fit into 192 bits
     */
    function toInt192(int256 value) internal pure returns (int192 downcasted) {
        downcasted = int192(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(192, value);
        }
    }

    /**
     * @dev Returns the downcasted int184 from int256, reverting on
     * overflow (when the input is less than smallest int184 or
     * greater than largest int184).
     *
     * Counterpart to Solidity's `int184` operator.
     *
     * Requirements:
     *
     * - input must fit into 184 bits
     */
    function toInt184(int256 value) internal pure returns (int184 downcasted) {
        downcasted = int184(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(184, value);
        }
    }

    /**
     * @dev Returns the downcasted int176 from int256, reverting on
     * overflow (when the input is less than smallest int176 or
     * greater than largest int176).
```

```
     *
     * Counterpart to Solidity's `int176` operator.
     *
     * Requirements:
     *
     * - input must fit into 176 bits
     */
    function toInt176(int256 value) internal pure returns (int176 downcasted) {
        downcasted = int176(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(176, value);
        }
    }

    /**
     * @dev Returns the downcasted int168 from int256, reverting on
     * overflow (when the input is less than smallest int168 or
     * greater than largest int168).
     *
     * Counterpart to Solidity's `int168` operator.
     *
     * Requirements:
     *
     * - input must fit into 168 bits
     */
    function toInt168(int256 value) internal pure returns (int168 downcasted) {
        downcasted = int168(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(168, value);
        }
    }

    /**
     * @dev Returns the downcasted int160 from int256, reverting on
     * overflow (when the input is less than smallest int160 or
     * greater than largest int160).
     *
     * Counterpart to Solidity's `int160` operator.
     *
     * Requirements:
     *
     * - input must fit into 160 bits
     */
    function toInt160(int256 value) internal pure returns (int160 downcasted) {
        downcasted = int160(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(160, value);
        }
    }

    /**
     * @dev Returns the downcasted int152 from int256, reverting on
     * overflow (when the input is less than smallest int152 or
     * greater than largest int152).
     *
     * Counterpart to Solidity's `int152` operator.
     *
     * Requirements:
     *
     * - input must fit into 152 bits
     */
    function toInt152(int256 value) internal pure returns (int152 downcasted) {
        downcasted = int152(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(152, value);
        }
```

```
    }

    /**
     * @dev Returns the downcasted int144 from int256, reverting on
     * overflow (when the input is less than smallest int144 or
     * greater than largest int144).
     *
     * Counterpart to Solidity's `int144` operator.
     *
     * Requirements:
     *
     * - input must fit into 144 bits
     */
    function toInt144(int256 value) internal pure returns (int144 downcasted) {
        downcasted = int144(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(144, value);
        }
    }

    /**
     * @dev Returns the downcasted int136 from int256, reverting on
     * overflow (when the input is less than smallest int136 or
     * greater than largest int136).
     *
     * Counterpart to Solidity's `int136` operator.
     *
     * Requirements:
     *
     * - input must fit into 136 bits
     */
    function toInt136(int256 value) internal pure returns (int136 downcasted) {
        downcasted = int136(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(136, value);
        }
    }

    /**
     * @dev Returns the downcasted int128 from int256, reverting on
     * overflow (when the input is less than smallest int128 or
     * greater than largest int128).
     *
     * Counterpart to Solidity's `int128` operator.
     *
     * Requirements:
     *
     * - input must fit into 128 bits
     */
    function toInt128(int256 value) internal pure returns (int128 downcasted) {
        downcasted = int128(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(128, value);
        }
    }

    /**
     * @dev Returns the downcasted int120 from int256, reverting on
     * overflow (when the input is less than smallest int120 or
     * greater than largest int120).
     *
     * Counterpart to Solidity's `int120` operator.
     *
     * Requirements:
     *
     * - input must fit into 120 bits
```

```
     */
    function toInt120(int256 value) internal pure returns (int120 downcasted) {
        downcasted = int120(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(120, value);
        }
    }

    /**
     * @dev Returns the downcasted int112 from int256, reverting on
     * overflow (when the input is less than smallest int112 or
     * greater than largest int112).
     *
     * Counterpart to Solidity's `int112` operator.
     *
     * Requirements:
     *
     * - input must fit into 112 bits
     */
    function toInt112(int256 value) internal pure returns (int112 downcasted) {
        downcasted = int112(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(112, value);
        }
    }

    /**
     * @dev Returns the downcasted int104 from int256, reverting on
     * overflow (when the input is less than smallest int104 or
     * greater than largest int104).
     *
     * Counterpart to Solidity's `int104` operator.
     *
     * Requirements:
     *
     * - input must fit into 104 bits
     */
    function toInt104(int256 value) internal pure returns (int104 downcasted) {
        downcasted = int104(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(104, value);
        }
    }

    /**
     * @dev Returns the downcasted int96 from int256, reverting on
     * overflow (when the input is less than smallest int96 or
     * greater than largest int96).
     *
     * Counterpart to Solidity's `int96` operator.
     *
     * Requirements:
     *
     * - input must fit into 96 bits
     */
    function toInt96(int256 value) internal pure returns (int96 downcasted) {
        downcasted = int96(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(96, value);
        }
    }

    /**
     * @dev Returns the downcasted int88 from int256, reverting on
     * overflow (when the input is less than smallest int88 or
     * greater than largest int88).
```

```
     *
     * Counterpart to Solidity's `int88` operator.
     *
     * Requirements:
     *
     * - input must fit into 88 bits
     */
    function toInt88(int256 value) internal pure returns (int88 downcasted) {
        downcasted = int88(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(88, value);
        }
    }

    /**
     * @dev Returns the downcasted int80 from int256, reverting on
     * overflow (when the input is less than smallest int80 or
     * greater than largest int80).
     *
     * Counterpart to Solidity's `int80` operator.
     *
     * Requirements:
     *
     * - input must fit into 80 bits
     */
    function toInt80(int256 value) internal pure returns (int80 downcasted) {
        downcasted = int80(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(80, value);
        }
    }

    /**
     * @dev Returns the downcasted int72 from int256, reverting on
     * overflow (when the input is less than smallest int72 or
     * greater than largest int72).
     *
     * Counterpart to Solidity's `int72` operator.
     *
     * Requirements:
     *
     * - input must fit into 72 bits
     */
    function toInt72(int256 value) internal pure returns (int72 downcasted) {
        downcasted = int72(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(72, value);
        }
    }

    /**
     * @dev Returns the downcasted int64 from int256, reverting on
     * overflow (when the input is less than smallest int64 or
     * greater than largest int64).
     *
     * Counterpart to Solidity's `int64` operator.
     *
     * Requirements:
     *
     * - input must fit into 64 bits
     */
    function toInt64(int256 value) internal pure returns (int64 downcasted) {
        downcasted = int64(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(64, value);
        }
```

```
    }

    /**
     * @dev Returns the downcasted int56 from int256, reverting on
     * overflow (when the input is less than smallest int56 or
     * greater than largest int56).
     *
     * Counterpart to Solidity's `int56` operator.
     *
     * Requirements:
     *
     * - input must fit into 56 bits
     */
    function toInt56(int256 value) internal pure returns (int56 downcasted) {
        downcasted = int56(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(56, value);
        }
    }

    /**
     * @dev Returns the downcasted int48 from int256, reverting on
     * overflow (when the input is less than smallest int48 or
     * greater than largest int48).
     *
     * Counterpart to Solidity's `int48` operator.
     *
     * Requirements:
     *
     * - input must fit into 48 bits
     */
    function toInt48(int256 value) internal pure returns (int48 downcasted) {
        downcasted = int48(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(48, value);
        }
    }

    /**
     * @dev Returns the downcasted int40 from int256, reverting on
     * overflow (when the input is less than smallest int40 or
     * greater than largest int40).
     *
     * Counterpart to Solidity's `int40` operator.
     *
     * Requirements:
     *
     * - input must fit into 40 bits
     */
    function toInt40(int256 value) internal pure returns (int40 downcasted) {
        downcasted = int40(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(40, value);
        }
    }

    /**
     * @dev Returns the downcasted int32 from int256, reverting on
     * overflow (when the input is less than smallest int32 or
     * greater than largest int32).
     *
     * Counterpart to Solidity's `int32` operator.
     *
     * Requirements:
     *
     * - input must fit into 32 bits
```

```
     */
    function toInt32(int256 value) internal pure returns (int32 downcasted) {
        downcasted = int32(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(32, value);
        }
    }

    /**
     * @dev Returns the downcasted int24 from int256, reverting on
     * overflow (when the input is less than smallest int24 or
     * greater than largest int24).
     *
     * Counterpart to Solidity's `int24` operator.
     *
     * Requirements:
     *
     * - input must fit into 24 bits
     */
    function toInt24(int256 value) internal pure returns (int24 downcasted) {
        downcasted = int24(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(24, value);
        }
    }

    /**
     * @dev Returns the downcasted int16 from int256, reverting on
     * overflow (when the input is less than smallest int16 or
     * greater than largest int16).
     *
     * Counterpart to Solidity's `int16` operator.
     *
     * Requirements:
     *
     * - input must fit into 16 bits
     */
    function toInt16(int256 value) internal pure returns (int16 downcasted) {
        downcasted = int16(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(16, value);
        }
    }

    /**
     * @dev Returns the downcasted int8 from int256, reverting on
     * overflow (when the input is less than smallest int8 or
     * greater than largest int8).
     *
     * Counterpart to Solidity's `int8` operator.
     *
     * Requirements:
     *
     * - input must fit into 8 bits
     */
    function toInt8(int256 value) internal pure returns (int8 downcasted) {
        downcasted = int8(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(8, value);
        }
    }

    /**
     * @dev Converts an unsigned uint256 into a signed int256.
     *
     * Requirements:
```

```
     *
     * - input must be less than or equal to maxInt256.
     */
    function toInt256(uint256 value) internal pure returns (int256) {
        // Note: Unsafe cast below is okay because `type(int256).max` is guaranteed to be positive
        if (value > uint256(type(int256).max)) {
            revert SafeCastOverflowedUintToInt(value);
        }
        return int256(value);
    }

    /**
     * @dev Cast a boolean (false or true) to a uint256 (0 or 1) with no jump.
     */
    function toUint(bool b) internal pure returns (uint256 u) {
        /// @solidity memory-safe-assembly
        assembly {
            u := iszero(iszero(b))
        }
    }
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.0.0) (utils/math/Math.sol)

pragma solidity ^0.8.20;

// Since Solidity 0.8, SafeMath is no longer needed
// Keep it here for readability, so we don't have to change the import paths in the rest of the code
library SafeMath {
    function add(uint256 x, uint256 y) internal pure returns (uint256 z) {
        z = x + y;
    }

    function sub(uint256 x, uint256 y) internal pure returns (uint256 z) {
        z = x - y;
    }

    function mul(uint256 x, uint256 y) internal pure returns (uint256 z) {
        z = x * y;
    }
}


// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.17;

import "./interfaces/ISwapERC20.sol";
import "./libraries/SafeMath.sol";

contract SwapERC20 is ISwapERC20 {
    string public constant name = "LinkSwap LP Token";
    string public constant symbol = "LSLP";
    uint8 public constant decimals = 18;
    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    bytes32 public DOMAIN_SEPARATOR;
    // keccak256("Permit(address owner,address spender,uint256 value,uint256 nonce,uint256 deadline)"
    bytes32 public constant PERMIT_TYPEHASH = 0x6e71edae12b1b97f4d1f60370fef10105fa2faae0126114a169c6
    mapping(address => uint256) public nonces;

    constructor() {
        uint256 chainId;
```

```
        assembly {
            chainId := chainid()
        }
        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                keccak256("EIP712Domain(string name,string version,uint256 chainId,address verifyingC
                keccak256(bytes(name)),
                keccak256(bytes("1")),
                chainId,
                address(this)
            )
        );
    }

    function _mint(address to, uint256 value) internal {
        totalSupply = totalSupply + value;
        balanceOf[to] = balanceOf[to] + value;
        emit Transfer(address(0), to, value);
    }

    function _burn(address from, uint256 value) internal {
        balanceOf[from] = balanceOf[from] - value;
        totalSupply = totalSupply - value;
        emit Transfer(from, address(0), value);
    }

    function _approve(address owner, address spender, uint256 value) private {
        allowance[owner][spender] = value;
        emit Approval(owner, spender, value);
    }

    function _transfer(address from, address to, uint256 value) private {
        balanceOf[from] = balanceOf[from] - value;
        balanceOf[to] = balanceOf[to] + value;
        emit Transfer(from, to, value);
    }

    function approve(address spender, uint256 value) external returns (bool) {
        _approve(msg.sender, spender, value);
        return true;
    }

    function transfer(address to, uint256 value) external returns (bool) {
        _transfer(msg.sender, to, value);
        return true;
    }

    function transferFrom(address from, address to, uint256 value) external returns (bool) {
        require(allowance[from][msg.sender] >= value, "INSUFFICIENT_ALLOWANCE");
        if (allowance[from][msg.sender] != type(uint256).max) {
            allowance[from][msg.sender] = allowance[from][msg.sender] - value;
        }
        _transfer(from, to, value);
        return true;
    }

    function permit(address owner, address spender, uint256 value, uint256 deadline, uint8 v, bytes32
        external
    {
        require(deadline >= block.timestamp, "EXPIRED");
        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                DOMAIN_SEPARATOR,
                keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, nonces[owner]++, deadlin
            )
```

```
        );
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(recoveredAddress != address(0) && recoveredAddress == owner, "INVALID_SIGNATURE");
        _approve(owner, spender, value);
    }
}


// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.17;

import "./SwapERC20.sol";
import "./SwapPair.sol";
import "./interfaces/ISwapFactory.sol";
import "./interfaces/ISwapPair.sol";

contract SwapFactory is ISwapFactory {
    address public owner;
    address public feeReceipt;

    mapping(address => mapping(address => address)) public getPair;
    address[] public allPairs;

    modifier onlyOwner() {
        require(msg.sender == owner, "SwapFactory: FORBIDDEN");
        _;
    }

    constructor(address _owner) {
        owner = _owner;
        feeReceipt = _owner;
    }

    function allPairsLength() external view returns (uint256) {
        return allPairs.length;
    }

    function createPair(address tokenA, address tokenB) external returns (address pair) {
        require(tokenA != tokenB, "Swap: IDENTICAL_ADDRESSES");
        (address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
        require(token0 != address(0), "Swap: ZERO_ADDRESS");
        require(getPair[token0][token1] == address(0), "Swap: PAIR_EXISTS"); // single check is suffi
        bytes memory bytecode = type(SwapPair).creationCode;
        bytes32 salt = keccak256(abi.encodePacked(token0, token1));
        assembly {
            pair := create2(0, add(bytecode, 32), mload(bytecode), salt)
        }
        ISwapPair(pair).initialize(token0, token1);
        getPair[token0][token1] = pair;
        getPair[token1][token0] = pair; // populate mapping in the reverse direction
        allPairs.push(pair);
        emit PairCreated(token0, token1, pair, allPairs.length);
    }

    function setFeeReceipt(address _feeReceipt) external onlyOwner {
        feeReceipt = _feeReceipt;
    }
}


// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.17;

import "./interfaces/ISwapERC20.sol";
import "./interfaces/ISwapPairBase.sol";
import "./SwapERC20.sol";
```

```
import "./libraries/Math.sol";
import "./libraries/core/UQ112x112.sol";
import "./interfaces/IERC20.sol";
import "./interfaces/ISwapFactory.sol";
import {ISwapV2Callee} from "./interfaces/ISwapCallee.sol";

contract SwapPair is ISwapPairBase, SwapERC20 {
    using SafeMath for uint256;
    using UQ112x112 for uint224;

    uint256 private constant DEFAULT_SWAP_FEE_RATE = 3;
    uint256 public constant MINIMUM_LIQUIDITY = 10 ** 3;
    bytes4 private constant SELECTOR = bytes4(keccak256(bytes("transfer(address,uint256)")));

    address public factory;
    address public token0;
    address public token1;

    uint112 private reserve0; // uses single storage slot, accessible via getReserves
    uint112 private reserve1; // uses single storage slot, accessible via getReserves
    uint32 private blockTimestampLast; // uses single storage slot, accessible via getReserves

    uint256 public price0CumulativeLast;
    uint256 public price1CumulativeLast;
    uint256 public kLast; // reserve0 * reserve1, as of immediately after the most recent liquidity e

    uint256 private unlocked = 1;

    // swap fees in x/1000 units
    uint256 public swapFeeRate;
    address public swapFeeReceipt;

    /// @dev Prevents calling a function from anyone except the address returned by ISwapFactory#owne
    modifier onlyFactoryOwner() {
        require(msg.sender == ISwapFactory(factory).owner(), "FORBIDDEN");
        _;
    }

    modifier lock() {
        require(unlocked == 1, "Swap: LOCKED");
        unlocked = 0;
        _;
        unlocked = 1;
    }

    constructor() {
        factory = msg.sender;
        swapFeeRate = DEFAULT_SWAP_FEE_RATE;
        swapFeeReceipt = ISwapFactory(factory).feeReceipt();
    }

    function getReserves() public view returns (uint112 _reserve0, uint112 _reserve1, uint32 _blockTi
        _reserve0 = reserve0;
        _reserve1 = reserve1;
        _blockTimestampLast = blockTimestampLast;
    }

    function _safeTransfer(address token, address to, uint256 value) private {
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(SELECTOR, to, value));
        require(success && (data.length == 0 || abi.decode(data, (bool))), "Swap: TRANSFER_FAILED");
    }

    // called once by the factory at time of deployment
    function initialize(address _token0, address _token1) external {
        require(msg.sender == factory, "Swap: FORBIDDEN"); // sufficient check
        token0 = _token0;
```

```
        token1 = _token1;
    }

    // update reserves and, on the first call per block, price accumulators
    function _update(uint256 balance0, uint256 balance1, uint112 _reserve0, uint112 _reserve1) privat
        require(balance0 <= type(uint112).max && balance1 <= type(uint112).max, "Swap: OVERFLOW");
        uint32 blockTimestamp = uint32(block.timestamp % 2 ** 32);
        uint32 timeElapsed = blockTimestamp - blockTimestampLast; // overflow is desired
        if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
            // * never overflows, and + overflow is desired
            price0CumulativeLast += uint256(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) * timeElaps
            price1CumulativeLast += uint256(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) * timeElaps
        }
        reserve0 = uint112(balance0);
        reserve1 = uint112(balance1);
        blockTimestampLast = blockTimestamp;
        emit Sync(reserve0, reserve1);
    }

    // this low-level function should be called from a contract which performs important safety check
    function mint(address to) external lock returns (uint256 liquidity) {
        (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
        uint256 balance0 = IERC20(token0).balanceOf(address(this));
        uint256 balance1 = IERC20(token1).balanceOf(address(this));
        uint256 amount0 = balance0.sub(_reserve0);
        uint256 amount1 = balance1.sub(_reserve1);

        uint256 _totalSupply = totalSupply;
        if (_totalSupply == 0) {
            liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
            _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tok
        } else {
            liquidity = Math.min(amount0.mul(_totalSupply) / _reserve0, amount1.mul(_totalSupply) / _
        }
        require(liquidity > 0, "Swap: INSUFFICIENT_LIQUIDITY_MINTED");
        _mint(to, liquidity);

        _update(balance0, balance1, _reserve0, _reserve1);
        emit Mint(msg.sender, amount0, amount1);
    }

    // this low-level function should be called from a contract which performs important safety check
    function burn(address to) external lock returns (uint256 amount0, uint256 amount1) {
        (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
        address _token0 = token0; // gas savings
        address _token1 = token1; // gas savings
        uint256 balance0 = IERC20(_token0).balanceOf(address(this));
        uint256 balance1 = IERC20(_token1).balanceOf(address(this));
        uint256 liquidity = balanceOf[address(this)];

        uint256 _totalSupply = totalSupply;
        amount0 = liquidity.mul(balance0) / _totalSupply; // using balances ensures pro-rata distribu
        amount1 = liquidity.mul(balance1) / _totalSupply; // using balances ensures pro-rata distribu
        require(amount0 > 0 && amount1 > 0, "Swap: INSUFFICIENT_LIQUIDITY_BURNED");
        _burn(address(this), liquidity);
        _safeTransfer(_token0, to, amount0);
        _safeTransfer(_token1, to, amount1);
        balance0 = IERC20(_token0).balanceOf(address(this));
        balance1 = IERC20(_token1).balanceOf(address(this));

        _update(balance0, balance1, _reserve0, _reserve1);
        emit Burn(msg.sender, amount0, amount1, to);
    }

    // this low-level function should be called from a contract which performs important safety check
    function swap(uint256 amount0Out, uint256 amount1Out, address to, bytes calldata data) external l
```

```
        require(amount0Out > 0 || amount1Out > 0, "Swap: INSUFFICIENT_OUTPUT_AMOUNT");
        (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
        require(amount0Out < _reserve0 && amount1Out < _reserve1, "Swap: INSUFFICIENT_LIQUIDITY");

        uint256 balance0;
        uint256 balance1;
        {
            // scope for _token{0,1}, avoids stack too deep errors
            address _token0 = token0;
            address _token1 = token1;
            require(to != _token0 && to != _token1, "Swap: INVALID_TO");
            if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // optimistically transfer to
            if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // optimistically transfer to
            if (data.length > 0) ISwapV2Callee(to).swapV2Call(msg.sender, amount0Out, amount1Out, dat
            balance0 = IERC20(_token0).balanceOf(address(this));
            balance1 = IERC20(_token1).balanceOf(address(this));
        }
        uint256 amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 - amount0Out) :
        uint256 amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 - amount1Out) :
        require(amount0In > 0 || amount1In > 0, "Swap: INSUFFICIENT_INPUT_AMOUNT");
        {
            // scope for reserve{0,1}Adjusted, avoids stack too deep errors
            uint256 balance0Adjusted = balance0.mul(1000).sub(amount0In.mul(3 + swapFeeRate));
            uint256 balance1Adjusted = balance1.mul(1000).sub(amount1In.mul(3 + swapFeeRate));
            require(
                balance0Adjusted.mul(balance1Adjusted) >= uint256(_reserve0).mul(_reserve1).mul(1000
            );
        }

        // collect swap fees
        if (amount0In > 0) {
            // fee0
            uint256 swapFee0 = amount0In / 1000 * swapFeeRate;
            balance0 -= swapFee0;
            _safeTransfer(token0, swapFeeReceipt, swapFee0);
        }

        if (amount1In > 0) {
            // fee1
            uint256 swapFee1 = amount1In / 1000 * swapFeeRate;
            balance1 -= swapFee1;
            _safeTransfer(token1, swapFeeReceipt, swapFee1);
        }

        _update(balance0, balance1, _reserve0, _reserve1);
        emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
    }

    // force balances to match reserves
    function skim(address to) external lock {
        address _token0 = token0; // gas savings
        address _token1 = token1; // gas savings
        _safeTransfer(_token0, to, IERC20(_token0).balanceOf(address(this)).sub(reserve0));
        _safeTransfer(_token1, to, IERC20(_token1).balanceOf(address(this)).sub(reserve1));
    }

    // force reserves to match balances
    function sync() external lock {
        _update(IERC20(token0).balanceOf(address(this)), IERC20(token1).balanceOf(address(this)), res
    }

    function setSwapFeeRate(uint256 _swapFeeRate) external onlyFactoryOwner {
        require(_swapFeeRate <= 200, "Swap: FEE_TOO_HIGH");
        swapFeeRate = _swapFeeRate;
    }
```

```
    function setSwapFeeReceipt(address _swapFeeReceipt) external onlyFactoryOwner {
        swapFeeReceipt = _swapFeeReceipt;
    }
}


// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.17;

import {ISwapRouter02} from "./interfaces/router/ISwapRouter02.sol";
import {ISwapFactory} from "./interfaces/ISwapFactory.sol";
import {ISwapPair} from "./interfaces/ISwapPair.sol";
import {TransferHelper} from "./libraries/router/TransferHelper.sol";
import {SwapRouterLibrary} from "./libraries/router/SwapRouterLibrary.sol";
import {IWNativeToken} from "./interfaces/router/IWNativeToken.sol";
import {IERC20} from "./interfaces/IERC20.sol";
import {SafeMath} from "./libraries/SafeMath.sol";

contract SwapRouter is ISwapRouter02 {
    using SafeMath for uint256;

    address private immutable _factory;
    address private immutable _WNativeToken;

    modifier ensure(uint256 deadline) {
        require(deadline >= block.timestamp, "SwapRouter: EXPIRED");
        _;
    }

    constructor(address factoryAddr, address wNativeTokenAddr) {
        _factory = factoryAddr;
        _WNativeToken = wNativeTokenAddr;
    }

    function factory() external view returns (address) {
        return _factory;
    }

    function WNativeToken() external view returns (address) {
        return _WNativeToken;
    }

    receive() external payable {
        assert(msg.sender == _WNativeToken); // only accept Native Token via fallback from the WNativ
    }

    // **** ADD LIQUIDITY ****
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint256 amountADesired,
        uint256 amountBDesired,
        uint256 amountAMin,
        uint256 amountBMin
    ) internal virtual returns (uint256 amountA, uint256 amountB) {
        // create the pair if it doesn't exist yet
        if (ISwapFactory(_factory).getPair(tokenA, tokenB) == address(0)) {
            ISwapFactory(_factory).createPair(tokenA, tokenB);
        }
        (uint256 reserveA, uint256 reserveB) = SwapRouterLibrary.getReserves(_factory, tokenA, tokenB
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint256 amountBOptimal = SwapRouterLibrary.quote(amountADesired, reserveA, reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(amountBOptimal >= amountBMin, "SwapRouter: INSUFFICIENT_B_AMOUNT");
```

```
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint256 amountAOptimal = SwapRouterLibrary.quote(amountBDesired, reserveB, reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, "SwapRouter: INSUFFICIENT_A_AMOUNT");
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }

    function addLiquidity(
        address tokenA,
        address tokenB,
        uint256 amountADesired,
        uint256 amountBDesired,
        uint256 amountAMin,
        uint256 amountBMin,
        address to,
        uint256 deadline
    ) external virtual override ensure(deadline) returns (uint256 amountA, uint256 amountB, uint256 l
        (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired, amountBDesired, amountAMin
        address pair = SwapRouterLibrary.pairFor(_factory, tokenA, tokenB);
        TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
        TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
        liquidity = ISwapPair(pair).mint(to);
    }

    function addLiquidityNativeToken(
        address token,
        uint256 amountTokenDesired,
        uint256 amountTokenMin,
        uint256 amountNativeTokenMin,
        address to,
        uint256 deadline
    )
        external
        payable
        virtual
        override
        ensure(deadline)
        returns (uint256 amountToken, uint256 amountNativeToken, uint256 liquidity)
    {
        (amountToken, amountNativeToken) =
            _addLiquidity(token, _WNativeToken, amountTokenDesired, msg.value, amountTokenMin, amount
        address pair = SwapRouterLibrary.pairFor(_factory, token, _WNativeToken);
        TransferHelper.safeTransferFrom(token, msg.sender, pair, amountToken);
        IWNativeToken(_WNativeToken).deposit{value: amountNativeToken}();
        assert(IWNativeToken(_WNativeToken).transfer(pair, amountNativeToken));
        liquidity = ISwapPair(pair).mint(to);
        // refund dust native token, if any
        if (msg.value > amountNativeToken) {
            TransferHelper.safeTransferNativeToken(msg.sender, msg.value - amountNativeToken);
        }
    }

    // **** REMOVE LIQUIDITY ****
    function removeLiquidity(
        address tokenA,
        address tokenB,
        uint256 liquidity,
        uint256 amountAMin,
        uint256 amountBMin,
        address to,
        uint256 deadline
    ) public virtual override ensure(deadline) returns (uint256 amountA, uint256 amountB) {
        address pair = SwapRouterLibrary.pairFor(_factory, tokenA, tokenB);
```

```
        ISwapPair(pair).transferFrom(msg.sender, pair, liquidity); // send liquidity to pair
        (uint256 amount0, uint256 amount1) = ISwapPair(pair).burn(to);
        (address token0,) = SwapRouterLibrary.sortTokens(tokenA, tokenB);
        (amountA, amountB) = tokenA == token0 ? (amount0, amount1) : (amount1, amount0);
        require(amountA >= amountAMin, "SwapRouter: INSUFFICIENT_A_AMOUNT");
        require(amountB >= amountBMin, "SwapRouter: INSUFFICIENT_B_AMOUNT");
    }

    function removeLiquidityNativeToken(
        address token,
        uint256 liquidity,
        uint256 amountTokenMin,
        uint256 amountNativeTokenMin,
        address to,
        uint256 deadline
    ) public virtual override ensure(deadline) returns (uint256 amountToken, uint256 amountNativeToke
        (amountToken, amountNativeToken) = removeLiquidity(
            token, _WNativeToken, liquidity, amountTokenMin, amountNativeTokenMin, address(this), dea
        );
        TransferHelper.safeTransfer(token, to, amountToken);
        IWNativeToken(_WNativeToken).withdraw(amountNativeToken);
        TransferHelper.safeTransferNativeToken(to, amountNativeToken);
    }

    function removeLiquidityWithPermit(
        address tokenA,
        address tokenB,
        uint256 liquidity,
        uint256 amountAMin,
        uint256 amountBMin,
        address to,
        uint256 deadline,
        bool approveMax,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external virtual override returns (uint256 amountA, uint256 amountB) {
        address pair = SwapRouterLibrary.pairFor(_factory, tokenA, tokenB);
        uint256 value = approveMax ? type(uint256).max : liquidity;
        ISwapPair(pair).permit(msg.sender, address(this), value, deadline, v, r, s);
        (amountA, amountB) = removeLiquidity(tokenA, tokenB, liquidity, amountAMin, amountBMin, to, d
    }

    function removeLiquidityNativeTokenWithPermit(
        address token,
        uint256 liquidity,
        uint256 amountTokenMin,
        uint256 amountNativeTokenMin,
        address to,
        uint256 deadline,
        bool approveMax,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external virtual override returns (uint256 amountToken, uint256 amountNativeToken) {
        address pair = SwapRouterLibrary.pairFor(_factory, token, _WNativeToken);
        uint256 value = approveMax ? type(uint256).max : liquidity;
        ISwapPair(pair).permit(msg.sender, address(this), value, deadline, v, r, s);
        (amountToken, amountNativeToken) =
            removeLiquidityNativeToken(token, liquidity, amountTokenMin, amountNativeTokenMin, to, de
    }

    // **** REMOVE LIQUIDITY (supporting fee-on-transfer tokens) ****

    function removeLiquidityNativeTokenSupportingFeeOnTransferTokens(
        address token,
```

```
        uint256 liquidity,
        uint256 amountTokenMin,
        uint256 amountNativeTokenMin,
        address to,
        uint256 deadline
    ) public virtual override ensure(deadline) returns (uint256 amountNativeToken) {
        (, amountNativeToken) = removeLiquidity(
            token, _WNativeToken, liquidity, amountTokenMin, amountNativeTokenMin, address(this), dea
        );
        TransferHelper.safeTransfer(token, to, IERC20(token).balanceOf(address(this)));
        IWNativeToken(_WNativeToken).withdraw(amountNativeToken);
        TransferHelper.safeTransferNativeToken(to, amountNativeToken);
    }

    function removeLiquidityNativeTokenWithPermitSupportingFeeOnTransferTokens(
        address token,
        uint256 liquidity,
        uint256 amountTokenMin,
        uint256 amountNativeTokenMin,
        address to,
        uint256 deadline,
        bool approveMax,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external virtual override returns (uint256 amountNativeToken) {
        address pair = SwapRouterLibrary.pairFor(_factory, token, _WNativeToken);
        uint256 value = approveMax ? type(uint256).max : liquidity;
        ISwapPair(pair).permit(msg.sender, address(this), value, deadline, v, r, s);
        amountNativeToken = removeLiquidityNativeTokenSupportingFeeOnTransferTokens(
            token, liquidity, amountTokenMin, amountNativeTokenMin, to, deadline
        );
    }

    // **** SWAP ****
    // requires the initial amount to have already been sent to the first pair

    function _swap(uint256[] memory amounts, address[] memory path, address _to) internal virtual {
        for (uint256 i; i < path.length - 1; i++) {
            (address input, address output) = (path[i], path[i + 1]);
            (address token0,) = SwapRouterLibrary.sortTokens(input, output);
            uint256 amountOut = amounts[i + 1];
            (uint256 amount0Out, uint256 amount1Out) =
                input == token0 ? (uint256(0), amountOut) : (amountOut, uint256(0));
            address to = i < path.length - 2 ? SwapRouterLibrary.pairFor(_factory, output, path[i + 2
            ISwapPair(SwapRouterLibrary.pairFor(_factory, input, output)).swap(amount0Out, amount1Out
        }
    }

    function swapExactTokensForTokens(
        uint256 amountIn,
        uint256 amountOutMin,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external virtual override ensure(deadline) returns (uint256[] memory amounts) {
        amounts = SwapRouterLibrary.getAmountsOut(_factory, amountIn, path);
        require(amounts[amounts.length - 1] >= amountOutMin, "SwapRouter: INSUFFICIENT_OUTPUT_AMOUNT"
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, SwapRouterLibrary.pairFor(_factory, path[0], path[1]), amounts[0]
        );
        _swap(amounts, path, to);
    }

    function swapTokensForExactTokens(
        uint256 amountOut,
```

```
        uint256 amountInMax,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external virtual override ensure(deadline) returns (uint256[] memory amounts) {
        amounts = SwapRouterLibrary.getAmountsIn(_factory, amountOut, path);
        require(amounts[0] <= amountInMax, "SwapRouter: EXCESSIVE_INPUT_AMOUNT");
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, SwapRouterLibrary.pairFor(_factory, path[0], path[1]), amounts[0]
        );
        _swap(amounts, path, to);
    }

    function swapExactNativeTokenForTokens(uint256 amountOutMin, address[] calldata path, address to,
        external
        payable
        virtual
        override
        ensure(deadline)
        returns (uint256[] memory amounts)
    {
        require(path[0] == _WNativeToken, "SwapRouter: INVALID_PATH");
        amounts = SwapRouterLibrary.getAmountsOut(_factory, msg.value, path);
        require(amounts[amounts.length - 1] >= amountOutMin, "SwapRouter: INSUFFICIENT_OUTPUT_AMOUNT"
        IWNativeToken(_WNativeToken).deposit{value: amounts[0]}();
        assert(IWNativeToken(_WNativeToken).transfer(SwapRouterLibrary.pairFor(_factory, path[0], pat
        _swap(amounts, path, to);
    }

    function swapTokensForExactNativeToken(
        uint256 amountOut,
        uint256 amountInMax,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external virtual override ensure(deadline) returns (uint256[] memory amounts) {
        require(path[path.length - 1] == _WNativeToken, "SwapRouter: INVALID_PATH");
        amounts = SwapRouterLibrary.getAmountsIn(_factory, amountOut, path);
        require(amounts[0] <= amountInMax, "SwapRouter: EXCESSIVE_INPUT_AMOUNT");
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, SwapRouterLibrary.pairFor(_factory, path[0], path[1]), amounts[0]
        );
        _swap(amounts, path, address(this));
        IWNativeToken(_WNativeToken).withdraw(amounts[amounts.length - 1]);
        TransferHelper.safeTransferNativeToken(to, amounts[amounts.length - 1]);
    }

    function swapExactTokensForNativeToken(
        uint256 amountIn,
        uint256 amountOutMin,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external virtual override ensure(deadline) returns (uint256[] memory amounts) {
        require(path[path.length - 1] == _WNativeToken, "SwapRouter: INVALID_PATH");
        amounts = SwapRouterLibrary.getAmountsOut(_factory, amountIn, path);
        require(amounts[amounts.length - 1] >= amountOutMin, "SwapRouter: INSUFFICIENT_OUTPUT_AMOUNT"
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, SwapRouterLibrary.pairFor(_factory, path[0], path[1]), amounts[0]
        );
        _swap(amounts, path, address(this));
        IWNativeToken(_WNativeToken).withdraw(amounts[amounts.length - 1]);
        TransferHelper.safeTransferNativeToken(to, amounts[amounts.length - 1]);
    }

    function swapNativeTokenForExactTokens(uint256 amountOut, address[] calldata path, address to, ui
```

```
        external
        payable
        virtual
        override
        ensure(deadline)
        returns (uint256[] memory amounts)
    {
        require(path[0] == _WNativeToken, "SwapRouter: INVALID_PATH");
        amounts = SwapRouterLibrary.getAmountsIn(_factory, amountOut, path);
        require(amounts[0] <= msg.value, "SwapRouter: EXCESSIVE_INPUT_AMOUNT");
        IWNativeToken(_WNativeToken).deposit{value: amounts[0]}();
        assert(IWNativeToken(_WNativeToken).transfer(SwapRouterLibrary.pairFor(_factory, path[0], pat
        _swap(amounts, path, to);
        // refund dust native token, if any
        if (msg.value > amounts[0]) TransferHelper.safeTransferNativeToken(msg.sender, msg.value - am
    }

    // **** SWAP (supporting fee-on-transfer tokens) ****
    // requires the initial amount to have already been sent to the first pair
    function _swapSupportingFeeOnTransferTokens(address[] memory path, address _to) internal virtual
        for (uint256 i; i < path.length - 1; i++) {
            (address input, address output) = (path[i], path[i + 1]);
            (address token0,) = SwapRouterLibrary.sortTokens(input, output);
            ISwapPair pair = ISwapPair(SwapRouterLibrary.pairFor(_factory, input, output));
            uint256 amountInput;
            uint256 amountOutput;
            {
                // scope to avoid stack too deep errors
                (uint256 reserve0, uint256 reserve1,) = pair.getReserves();
                (uint256 reserveInput, uint256 reserveOutput) =
                    input == token0 ? (reserve0, reserve1) : (reserve1, reserve0);
                amountInput = IERC20(input).balanceOf(address(pair)).sub(reserveInput);
                amountOutput = SwapRouterLibrary.getAmountOut(amountInput, reserveInput, reserveOutpu
            }
            (uint256 amount0Out, uint256 amount1Out) =
                input == token0 ? (uint256(0), amountOutput) : (amountOutput, uint256(0));
            address to = i < path.length - 2 ? SwapRouterLibrary.pairFor(_factory, output, path[i + 2
            pair.swap(amount0Out, amount1Out, to, new bytes(0));
        }
    }

    function swapExactTokensForTokensSupportingFeeOnTransferTokens(
        uint256 amountIn,
        uint256 amountOutMin,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external virtual override ensure(deadline) {
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, SwapRouterLibrary.pairFor(_factory, path[0], path[1]), amountIn
        );
        uint256 balanceBefore = IERC20(path[path.length - 1]).balanceOf(to);
        _swapSupportingFeeOnTransferTokens(path, to);
        require(
            IERC20(path[path.length - 1]).balanceOf(to).sub(balanceBefore) >= amountOutMin,
            "SwapRouter: INSUFFICIENT_OUTPUT_AMOUNT"
        );
    }

    function swapExactNativeTokenForTokensSupportingFeeOnTransferTokens(
        uint256 amountOutMin,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external payable virtual override ensure(deadline) {
        require(path[0] == _WNativeToken, "SwapRouter: INVALID_PATH");
```

```
        uint256 amountIn = msg.value;
        IWNativeToken(_WNativeToken).deposit{value: amountIn}();
        assert(IWNativeToken(_WNativeToken).transfer(SwapRouterLibrary.pairFor(_factory, path[0], pat
        uint256 balanceBefore = IERC20(path[path.length - 1]).balanceOf(to);
        _swapSupportingFeeOnTransferTokens(path, to);
        require(
            IERC20(path[path.length - 1]).balanceOf(to).sub(balanceBefore) >= amountOutMin,
            "SwapRouter: INSUFFICIENT_OUTPUT_AMOUNT"
        );
    }

    function swapExactTokensForNativeTokenSupportingFeeOnTransferTokens(
        uint256 amountIn,
        uint256 amountOutMin,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external virtual override ensure(deadline) {
        require(path[path.length - 1] == _WNativeToken, "SwapRouter: INVALID_PATH");
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, SwapRouterLibrary.pairFor(_factory, path[0], path[1]), amountIn
        );
        _swapSupportingFeeOnTransferTokens(path, address(this));
        uint256 amountOut = IERC20(_WNativeToken).balanceOf(address(this));
        require(amountOut >= amountOutMin, "SwapRouter: INSUFFICIENT_OUTPUT_AMOUNT");
        IWNativeToken(_WNativeToken).withdraw(amountOut);

        TransferHelper.safeTransferNativeToken(to, amountOut);
    }

    // **** LIBRARY FUNCTIONS ****
    function quote(uint256 amountA, uint256 reserveA, uint256 reserveB)
        public
        pure
        virtual
        override
        returns (uint256 amountB)
    {
        return SwapRouterLibrary.quote(amountA, reserveA, reserveB);
    }

    function getAmountOut(uint256 amountIn, uint256 reserveIn, uint256 reserveOut)
        public
        pure
        virtual
        override
        returns (uint256 amountOut)
    {
        return SwapRouterLibrary.getAmountOut(amountIn, reserveIn, reserveOut);
    }

    function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut)
        public
        pure
        virtual
        override
        returns (uint256 amountIn)
    {
        return SwapRouterLibrary.getAmountIn(amountOut, reserveIn, reserveOut);
    }

    function getAmountsOut(uint256 amountIn, address[] memory path)
        public
        view
        virtual
        override
```

```
        returns (uint256[] memory amounts)
    {
        return SwapRouterLibrary.getAmountsOut(_factory, amountIn, path);
    }

    function getAmountsIn(uint256 amountOut, address[] memory path)
        public
        view
        virtual
        override
        returns (uint256[] memory amounts)
    {
        return SwapRouterLibrary.getAmountsIn(_factory, amountOut, path);
    }
}
```

Analysis of audit results

Re-Entrancy

- **Description:**
  One of the features of smart contracts is the ability to call and utilise code of other external contracts. Contracts also typically handle Blockchain Currency, and as such often send Blockchain Currency to various external user addresses. The operation of calling external contracts, or sending Blockchain Currency to an address, requires the contract to submit an external call. These external calls can be hijacked by attackers whereby they force the contract to execute further code (i.e. through a fallback function) , including calls back into itself. Thus the code execution "re-enters" the contract. Attacks of this kind were used in the infamous DAO hack.
- **Detection results:**

  ```
  PASSED!
  ```

- **Security suggestion:**
  no.

Arithmetic Over/Under Flows

- **Description:**
  The Virtual Machine (EVM) specifies fixed-size data types for integers. This means that an integer variable, only has a certain range of numbers it can represent. A uint8 for example, can only store numbers in the range [0,255]. Trying to store 256 into a uint8 will result in 0. If care is not taken, variables in Solidity can be exploited if user input is unchecked and calculations are performed which result in numbers that lie outside the range of the data type that stores them.
- **Detection results:**

  ```
  PASSED!
  ```

- **Security suggestion:**
  no.

Unexpected Blockchain Currency

- **Description:**
  Typically when Blockchain Currency is sent to a contract, it must execute either the fallback function, or another function described in the contract. There are two exceptions to this, where Blockchain Currency can exist in a contract without having executed any code. Contracts which rely on code execution for every Blockchain Currency sent to the contract can be vulnerable to attacks where Blockchain Currency is forcibly sent to a contract.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Delegatecall

- **Description:**
  The CALL and DELEGATECALL opcodes are useful in allowing developers to modularise their code. Standard external message calls to contracts are handled by the CALL opcode whereby code is run in the context of the external contract/function. The DELEGATECALL opcode is identical to the standard message call, except that the code executed at the targeted address is run in the context of the calling contract along with the fact that msg.sender and msg.value remain unchanged. This feature enables the implementation of libraries whereby developers can create reusable code for future contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Default Visibilities

- **Description:**
  Functions in Solidity have visibility specifiers which dictate how functions are allowed to be called. The visibility determines whBlockchain Currency a function can be called externally by users, by other derived contracts, only internally or only externally. There are four visibility specifiers, which are described in detail in the Solidity Docs. Functions default to public allowing users to call them externally. Incorrect use of visibility specifiers can lead to some devestating vulernabilities in smart contracts as will be discussed in this section.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Entropy Illusion

- **Description:**
  All transactions on the blockchain are deterministic state transition operations. Meaning that every transaction modifies the global state of the ecosystem and it does so in a calculable way with no uncertainty. This ultimately means that inside the blockchain ecosystem there is no source of entropy or randomness. There is no rand() function in Solidity. Achieving decentralised entropy (randomness) is a well established problem and many

ideas have been proposed to address this (see for example, RandDAO or using a chain of Hashes as described by Vitalik in this post).

- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

External Contract Referencing

- **Description:**
  One of the benefits of the global computer is the ability to re-use code and interact with contracts already deployed on the network. As a result, a large number of contracts reference external contracts and in general operation use external message calls to interact with these contracts. These external message calls can mask malicious actors intentions in some non-obvious ways, which we will discuss.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unsolved TODO comments

- **Description:**
  Check for Unsolved TODO comments
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Short Address/Parameter Attack

- **Description:**
  This attack is not specifically performed on Solidity contracts themselves but on third party applications that may interact with them. I add this attack for completeness and to be aware of how parameters can be manipulated in contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unchecked CALL Return Values

- **Description:**
  There a number of ways of performing external calls in solidity. Sending Blockchain Currency to external accounts is commonly performed via the transfer() method. However, the send() function can also be used and, for more versatile external calls, the CALL opcode can be directly employed in solidity. The call() and send() functions return a boolean indicating if the call succeeded or failed. Thus these functions have a simple caveat, in that the transaction that executes these functions will not revert if the external call (intialised by call() or send()) fails, rather the call() or send() will simply return false. A common pitfall arises when the return value is not checked, rather the developer expects a revert to occur.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Race Conditions / Front Running

- **Description:**
  The combination of external calls to other contracts and the multi-user nature of the underlying blockchain gives rise to a variety of potential Solidity pitfalls whereby users race code execution to obtain unexpected states. Re-Entrancy is one example of such a race condition. In this section we will talk more generally about different kinds of race conditions that can occur on the blockchain. There is a variety of good posts on this subject, a few are: Wiki - Safety, DASP - Front-Running and the Consensus - Smart Contract Best Practices.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Denial Of Service (DOS)

- **Description:**
  This category is very broad, but fundamentally consists of attacks where users can leave the contract inoperable for a small period of time, or in some cases, permanently. This can trap Blockchain Currency in these contracts forever, as was the case with the Second Parity MultiSig hack
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Block Timestamp Manipulation

- **Description:**
  Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion section for further details), locking funds for periods of time and various state-changing conditional statements that are time-dependent. Miner's have the ability to adjust timestamps slightly which can prove to be quite dangerous if block timestamps are used incorrectly in smart contracts.

- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Constructors with Care

- **Description:**
  Constructors are special functions which often perform critical, privileged tasks when initialising contracts. Before solidity v0.4.22 constructors were defined as functions that had the same name as the contract that contained them. Thus, when a contract name gets changed in development, if the constructor name isn't changed, it becomes a normal, callable function. As you can imagine, this can (and has) lead to some interesting contract hacks.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unintialised Storage Pointers

- **Description:**
  The EVM stores data either as storage or as memory. Understanding exactly how this is done and the default types for local variables of functions is highly recommended when developing contracts. This is because it is possible to produce vulnerable contracts by inappropriately intialising variables.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Floating Points and Numerical Precision

- **Description:**
  As of this writing (Solidity v0.4.24), fixed point or floating point numbers are not supported. This means that floating point representations must be made with the integer types in Solidity. This can lead to errors/vulnerabilities if not implemented correctly.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

tx.origin Authentication

- **Description:**
  Solidity has a global variable, tx.origin which traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in smart contracts leaves the contract vulnerable to a phishing-like attack.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Permission restrictions

- **Description:**
  Contract managers who can control liquidity or pledge pools, etc., or impose unreasonable restrictions on other users.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

armors.io

contact@armors.io