Cyberscope

# Audit Report

# SMARTSWAP

November 2023

| | |
|---|---|
| Network | ETH |
| Address | 0x8eAe4006183B421ea22ba01B5697B6852d6C4916 |
| Network | BSC |
| Address | 0x7Fa5Fd33B053cE18678B6B862290A115F779c328 |
| Network | TRX |
| Address | TTeZFaYtxkyS2GUfToqr6TonBT7ed2t2JM |
| Network | ETH |
| Address | 0xd36F2c5E251A66B89dB3dB9536dE7DCa0c3f1433 |
| Network | BSC |
| Address | 0x836891159d630690a2652BD5864074bafD11eA3a |
| Network | TRX |
| Address | TLTNrWtXPTQ4hfQnYShEc6QqqXTUs4v8Mr |
| Audited by | © cyberscope |

# Table of Contents

# Review

| | |
|---|---|
| **ETH (DP2PCoin)** | https://etherscan.io/address/0x8eAe4006183B421ea22ba01B5697B6852d6C4916 |
| **BSC (DP2PCoin)** | https://bscscan.com/address/0x7Fa5Fd33B053cE18678B6B862290A115F779c328 |
| **TRX (DP2PCoin)** | https://tronscan.org/#/contract/TTeZFaYtxkyS2GUfToqr6TonBT7ed2t2JM |
| **ETH (DP2PToken)** | https://etherscan.io/address/0xd36F2c5E251A66B89dB3dB9536dE7DCa0c3f1433 |
| **BSC (DP2PToken)** | https://bscscan.com/address/0x836891159d630690a2652BD5864074bafD11eA3a |
| **TRX (DP2PTokenLegacyUSDT)** | https://tronscan.org/#/contract/TLTNrWtXPTQ4hfQnYShEc6QqqXTUs4v8Mr |

## Audit Updates

| | |
|---|---|
| **Initial Audit** | 28 Oct 2023 |

## Source Files

| Filename | SHA256 |
|---|---|
| **DP2PCoin.sol** | 4280298caa35a54a3f726e6745f534bec5288803238ad2711f169e4e5fc5b834 |
| **DP2PToken.sol** | ea4e2d1be10235812b0749f52b527367c515419d46b3cd2534061c853e774c37 |
| **DP2PTokenLegacyUSDT.sol** | 92be5ecd34083138db18211b3d5e709ca82fd8a8df09716bfc183c4f5b32535a |

# Overview

In the SMARTSWAP there is no outside influence on contracts.

The SMARTSWAP project is an intricate ecosystem of smart contracts segregated into three principal groups, each serving a distinct function within the realm of decentralized peer-to-peer transactions across various blockchain networks.

1. DP2PCoin Contracts:

Network Deployment: Ethereum (ETH), Binance Smart Chain (BNB), and TRON (TRX). Functionality: These contracts are tailored to facilitate and manage deals utilizing the native cryptocurrency of each respective blockchain network. They are designed to operate seamlessly with the inherent tokens—ETH on Ethereum, BNB on Binance Smart Chain, and TRX on TRON—enabling users to conduct transactions with the native blockchain assets through a decentralized framework.

2. DP2PToken Contracts:

Network Deployment: Ethereum (ETH) and Binance Smart Chain (BNB). Functionality: Deployed on networks that support the ERC20 token standard, these contracts extend the SMARTSWAP project's functionality to include deals conducted with ERC20 tokens. They provide a structure for the creation, management, and completion of deals, leveraging the flexibility of ERC20 tokens to engage a broader range of assets and potentially accommodate a variety of token-specific features and utilities.

3. DP2PTokenLegacyUSDT Contracts:

Network Deployment: TRON (TRX) network. Functionality: This specialized group consists of a single contract designed specifically for the TRON network, emphasizing deals made with the TRON-based iteration of the USDT token. The DP2PTokenLegacyUSDT contract is engineered to support the initiation, handling, and resolution of transactions using USDT, allowing users to benefit from the stability and widespread adoption of this particular stablecoin within the TRON ecosystem.

Each of the contracts within the SMARTSWAP project framework shares the same functional purpose—facilitating and managing peer-to-peer transactions. The distinguishing

factor between the contracts lies solely in the blockchain network on which they are deployed and the type of assets they are designed to handle.

## startDeal Functionality

The `startDeal` function within the SMARTSWAP project contracts serves as a gateway for users to initiate new peer-to-peer transactions. Upon invocation, a user submits a set of parameters defined by the `DealData` struct, which includes vital information such as the token's address, the seller and buyer's addresses, the arbiter, the receiver of the transaction fee, the amount of the token being dealt, the fee for the transaction, a nonce, and an extra bytes32 value for extended functionality.

The function commences by generating a unique hash from the provided `DealData` parameters, which acts as a distinctive identifier for the new deal. It ensures that this hash has not been used before to prevent any possible conflicts with existing deals. Once the uniqueness is verified, the function secures the assets by transferring the specified token amount and the fee from the seller to the contract's address. This action moves the assets into the contract, signifying that the deal has been set in motion.

Following the successful transfer, the contract updates the state of the deal to `START`, marking the official commencement of the transaction. Additionally, the `startDeal` function emits a `StateChanged` event, providing a timestamp on the blockchain of the deal's initiation, which includes the deal's hash, the submitted parameters, the new state, and the initiating sender's address.

## clearDealSeller Functionality

The `clearDealSeller` function provide sellers the mechanism to finalize and settle a deal. Upon invoking this function, the contract first confirms that the caller, or `msg.sender`, is indeed the seller associated with the deal in question. This verification is critical to ensure that only the authorized party can trigger the clearing of the deal. Once the seller's identity is authenticated, the function proceeds to check the existence and the current state of the deal to ensure it is eligible for clearing. Following these validations, the function is allocating the transaction fee ( `deal.fee` ) to the contract's cumulative `fees` map. Then, it proceeds the transfer of the `deal.amount` to the buyer.

The function also updates the state of the deal to `CLEARED_SELLER`, marking the successful conclusion of the transaction from the seller's end. This state change is broadcasted through a `StateChanged` event emission, which logs the new state along with the deal's unique identifier and the involved parties' addresses, thereby maintaining a transparent record on the blockchain.

## cancelDealBuyer Functionality

The `cancelDealBuyer` function empowers the buyer with the capability to terminate a deal. This function can only be initiated by the buyer, as confirmed by the `onlyBuyer` modifier which asserts that the `msg.sender` must match the buyer specified in the deal. This check is crucial to ensure that the cancellation is executed solely by the authorized individual. Upon activation, the function also verifies, through the `isValidStateTransfer` modifier, that the deal is currently in a state that permits cancellation by the buyer, specifically looking for the deal to be in the `START` state.

After the preliminary checks, the `cancelDealBuyer` function internally calls `_cancelDeal`, which is responsible for the actual cancellation process. In `_cancelDeal`, the funds, inclusive of the deal amount and the associated fee, are returned to the seller. This implies that the buyer relinquishes their claim to the deal's assets, and the seller receives back the full amount they were set to the deal.

In addition to the fund transfer, the deal's state is updated to `CANCELED_BUYER`, reflecting the deal's cancellation from the buyer's side. This change in the state of the deal is publicly logged on the blockchain through the emission of a `StateChanged` event. The event captures and broadcasts the deal's unique identifier, the hash of the deal data, the updated state to CANCELED_BUYER, and the address of the msg.sender (the buyer in this case), ensuring that there is a transparent and tamper-evident record of the transaction's cancellation.

## cancelTimeoutArbiter and cancelDealArbiter Functionality

The `cancelTimeoutArbiter` and `cancelDealArbiter` functions are arbitrator-initiated cancellations within the smart contract that share a similar process with a key distinction in the final state they set for the deal. Both functions are guarded by the `onlyArbiter` modifier, which restricts execution to the designated arbiter of the deal, ensuring that only an authorized arbitrator can invoke these cancellations. The

`isValidStateTransfer` modifier is also employed in both functions to ensure that the deal is in an appropriate state to transition to a cancellation state.

The primary difference between the two functions lies in the specific cancellation state that is set for the deal. In `cancelTimeoutArbiter`, the deal is moved to the `CANCELED_TIMEOUT_ARBITER` state, indicating that the arbitration process could not be completed within a predefined time frame. On the other hand, `cancelDealArbiter` sets the deal's state to `CANCELED_ARBITER`, which suggests that the arbiter has actively decided to cancel the deal.

Both functions ultimately call the `_cancelDeal` internal function, which handles the actual transfer of funds back to the seller, inclusive of the deal amount and the fee. This ensures that the funds are returned to the seller upon cancellation by the arbiter, regardless of the specific cancellation reason.

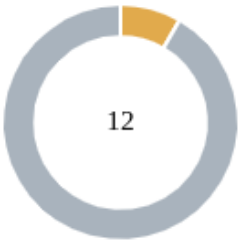
## clearDealArbiter Functionality

The `clearDealArbiter` function, is invoked to finalize and clear a deal, but only under the arbiter’s oversight. This function is guarded by `onlyArbiter` to ensure that the arbiter is the executor, and by `isValidStateTransfer` to validate the state transition to `CLEARED_ARBITER`. The `_clearDeal` internal function handles the distribution of the deal’s fee to the fee receiver's cumulative fees map and transfers the deal amount to the buyer. This resolution reflects that the arbiter has determined the buyer should receive the funds, and the fee is allocated as per the contract terms. The finalization of the deal is made public through the emission of a `StateChanged` event, which logs the deal’s update for transparency.

## claim Functionality

The `claim` function is designed for fee recipients to withdraw their earned fees from the contract. It can only be utilized by parties who have been assigned as fee recipients during the initial deal setup via the `startDeal` function and who have since accumulated fees. The function ensures that only those with a positive fee balance can claim; it checks that the claimant's accumulated fee for a given token is greater than zero. Upon verification, the accumulated fee is reset to zero to prevent reclamation, and the specified amount is transferred to the receiver's address, completing the claim process.

# Findings Breakdown




| | |
|---|---|
| Critical | 0 |
| Medium | 1 |
| Minor / Informative | 11 |

| Severity | Unresolved | Acknowledged | Resolved | Other |
|---|---|---|---|---|
| Critical | 0 | 0 | 0 | 0 |
| Medium | 1 | 0 | 0 | 0 |
| Minor / Informative | 11 | 0 | 0 | 0 |

# Diagnostics

Critical Medium Minor / Informative

| Severity | Code | Description | Status |
|---|---|---|---|
| Medium | PTAI | Potential Transfer Amount Inconsistency | Unresolved |
| Minor / Informative | CCF | Contradictory Contract Functionality | Unresolved |
| Minor / Informative | RPF | Redundant Payable Function | Unresolved |
| Minor / Informative | MTP | Misleading Token Parameter | Unresolved |
| Minor / Informative | CFGO | Cancel Functionality Gas Optimization | Unresolved |
| Minor / Informative | MEE | Missing Events Emission | Unresolved |
| Minor / Informative | RSV | Redundant Struct Variable | Unresolved |
| Minor / Informative | RED | Redundant Enum Declarations | Unresolved |
| Minor / Informative | L09 | Dead Code Elimination | Unresolved |
| Minor / Informative | L17 | Usage of Solidity Assembly | Unresolved |
| Minor / Informative | L18 | Multiple Pragma Directives | Unresolved |
| Minor / Informative | L19 | Stable Compiler Version | Unresolved |

## PTAI - Potential Transfer Amount Inconsistency

| | |
|---|---|
| **Criticality** | Medium |
| **Location** | DP2PToken.sol#L392,451 |
| **Status** | Unresolved |

## Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

| Tax | Amount | Expected | Actual |
|---|---|---|---|
| No Tax | 100 | 100 | 100 |
| 10% Tax | 100 | 100 | 90 |

As a result, the contract considers that an amount of `deal.amount + deal.fee` of the `deal.token` has been transferred to the contract, but the actual amount transferred could be less.

```
    function startDeal(
          DealData calldata deal
     ) external payable virtual override onlySeller(deal)
returns (bytes32) {
          ...
          _transferFrom(deal.token, deal.seller, address(this),
deal.amount + deal.fee);
          ...
     }

     function _cancelDeal(DealData calldata deal) internal {
          _transfer(deal.token, deal.seller, deal.amount +
deal.fee);
     }

     function _clearDeal(DealData calldata deal) internal {
          fees[deal.token][deal.feeReceiver] += deal.fee;
          _transfer(deal.token, deal.buyer, deal.amount);
     }
```

## Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
 Actual Transferred Amount = Balance After Transfer - Balance
Before Transfer
```

## CCF - Contradictory Contract Functionality

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | DP2PTokenLegacyUSDT.sol#L392,792 |
| **Status** | Unresolved |

## Description

The `DP2PTokenLegacyUSDT` contract is specifically designed to interact with the `USDT` (Tether) token for handling the deal functionalities. Despite this implication, the `startDeal` function within the contract does not enforce the use of the USDT token; it accepts any token address provided by the user as the `deal.token` parameter. This design allows for the possibility of initiating deals with any ERC20 token, not just `USDT`. The result is a discrepancy between the contract's name and its actual functionality, as it is capable of handling transactions with tokens other than `USDT`, which could mislead users and stakeholders.

```
    function startDeal(
        DealData calldata deal
    ) external payable virtual override onlySeller(deal) returns
(bytes32) {
        bytes32 _hash = dealHash(deal);
        ...

        _transferFrom(deal.token, deal.seller, address(this),
deal.amount + deal.fee);
        ...
    }

    contract DP2PTokenLegacyUSDT is DP2PStateMachine {
    ...
    }
```

## Recommendation

It is recommended to reconsider the contract's functionality to align with its naming convention. If the purpose is to handle deals exclusively with the USDT token, the contract should be modified to utilize a fixed USDT token address that cannot be altered by the user inputs. This could involve hardcoding the USDT token address into the contract or providing

it via a constructor or initializer that sets it as a constant or immutable state variable. By doing so, the contract will enforce its specific use case with USDT, preventing any ambiguity and ensuring that users interact with the contract as intended.

# RPF - Redundant Payable Function

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | DP2PToken.sol#L394<br>DP2PTokenLegacy<br>USDT.sol#L394 |
| **Status** | Unresolved |

## Description

The contract's function `startDeal` has a `payable` modifier which means a user pays with the native token. The function does not use the `msg.value` variable anywhere inside the function to indicate the usage of the `payable` modifier. As a result, the `payable` modifier is redundant.

```
   function startDeal(
        DealData calldata deal
    ) external payable virtual override onlySeller(deal)
returns (bytes32) {
        bytes32 _hash = dealHash(deal);
        require(deals[_hash] == DealState.ZERO, 'storage slot
collision');
        deals[_hash] = DealState.START;

        _transferFrom(deal.token, deal.seller, address(this),
deal.amount + deal.fee);

        emit StateChanged(_hash, deal, DealState.START,
msg.sender);
        return _hash;
    }
```

## Recommendation

The team is advised to remove the `payable` modifier from the function's declaration as it is not needed.

## MTP - Misleading Token Parameter

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | DP2PCoin.sol#L709 |
| **Status** | Unresolved |

### Description

The contract uses the `startDeal` function that accepts a `DealData` struct as an input parameter. This struct includes the variable `deal.token`. The `startDeal` function is designed to work with native tokens (e.g., Ethereum's ETH) given the use of `msg.value` to handle transactions. As such, the function does not utilize the `deal.token` variable when initiating a deal. The inclusion of `deal.token` in the parameters leads to a discrepancy between the function's intended use and its parameters, potentially causing confusion for users. They may assume that `deal.token` should be specified, even though it has no impact on the transaction, and could set this parameter when initiating a deal with native tokens.

```
    function startDeal(
        DealData calldata deal
    ) external payable override onlySeller(deal) returns (bytes32) {
        bytes32 _hash = dealHash(deal);
        require(deals[_hash] == DealState.ZERO, 'storage slot collision');
        require(msg.value > 0 && msg.value == deal.amount + deal.fee, 'wrong ether amount');
        deals[_hash] = DealState.START;

        emit StateChanged(_hash, deal, DealState.START, msg.sender);
        return _hash;
    }
```

### Recommendation

It is recommended to refactor the `startDeal` function to remove the `deal.token` parameter, as it is not applicable for transactions with native tokens. This will streamline the function interface, reducing confusion and preventing users from providing an unnecessary

and unused parameter. The refactored function should clearly indicate through its parameters and documentation that it is intended for use with native tokens only, thus eliminating any ambiguity regarding the purpose and usage of the `deal.token` variable. This change will enhance the contract's usability and ensure that the function's parameters are in full alignment with its logic.

# CFGO - Cancel Functionality Gas Optimization

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | DP2PCoin.sol#L631<br>Token.sol#L405<br>DP2PTokenLegacyUSDT.sol#L405 |
| **Status** | Unresolved |

## Description

The contract is designed to allow the `buyer` address of a deal to cancel them through the `cancelDealBuyer` function. This function, when invoked, transfers the assets back to the `seller` address of the deal. However, the cancelation process requires the `buyer` to pay the amount of gas needed for the transaction. This gas cost can be a deterrent for the `buyer` users, discouraging them from canceling deals even if they want to cancel them.

```
    function cancelDealBuyer(
        DealData calldata deal
    ) external override onlyBuyer(deal) isValidStateTransfer(deal,
DealState.CANCELED_BUYER) {
        _cancelDeal(deal);
    }

    function _cancelDeal(DealData calldata deal) internal {
        _transfer(deal.token, deal.seller, deal.amount + deal.fee);
    }
```

## Recommendation

It is recommended to only update the status of the deal, which allows the `buyer` address to indicate the intention to cancel the deal without incurring the gas costs of the actual asset transfers back to the `seller` address. The `seller` can then invoke a separate function to finalize the cancellation and bear the gas costs associated with the asset transfer. This approach ensures that the `buyer` is not discouraged from canceling deals due to high gas costs, and the `seller`, who benefits from the return of their assets, bears the associated costs.

# MEE - Missing Events Emission

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | DP2PCoin.sol#L670 |
| **Status** | Unresolved |

## Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
function claim(address token, address receiver) external override {
    ...
}
```

## Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

## RSV - Redundant Struct Variable

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | DP2PCoin.sol#L563<br>DP2PToken.sol#L328<br>DP2PTokenLegacyUSDT.sol#L328 |
| **Status** | Unresolved |

### Description

The contract is utilazin the `DealData` struct that encapsulates transaction-related information, including addresses for token, seller, buyer, arbiter, and feeReceiver, alongside values for amount, fee, and a nonce to ensure unique transaction hashes. The struct also declares the `bytes32` variable `extra`. However, `the` extra variable is not utilized in any of the contract's functions or state changes. As a result it does not contribute to the contract's logic or provide additional functionality. The presence of this non-functional variable leads to increased gas costs for operations involving the struct, as it occupies storage space without serving a purpose.

```
struct DealData {
        address token;
        address seller;
        address buyer;
        address arbiter;
        address feeReceiver;
        uint256 amount;
        uint256 fee;
        uint256 nonce;
        bytes32 extra;
    }
```

### Recommendation

It is recommended to remove the `bytes32 extra` variable from the `DealData` struct since it does not alter the contract logic or add any meaningful functionality. Eliminating this variable will result in gas optimization, reducing the cost associated with deploying and interacting with the contract. Furthermore, the removal of unnecessary

variables can simplify the contract's structure, making it more readable and easier to maintain.

## RED - Redundant Enum Declarations

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | DP2PCoin.sol#L541<br>DP2PToken.sol#L315<br>DP2PTokenLegacyUSDT.sol#L315 |
| **Status** | Unresolved |

### Description

The contract includes the `DealState` enum within the `IDP2PStateMachine` interface that declares various states, including `PAYMENT_COMPLETE`, `DISPUTE`, and `CANCELED_SELLER`. These specific states are not actively used in any part of the contract's functions or state transitions. As enums contribute to the contract's byte code, their presence without utilization leads to unnecessary gas consumption during contract deployment and can also cause confusion about the contract's actual state management logic.

```
    enum DealState {
        ZERO, // 0
        START, // 1
        PAYMENT_COMPLETE, // 2 legacy
        DISPUTE, // 3 legacy
        CANCELED_ARBITER, // 4
        CANCELED_TIMEOUT_ARBITER, // 5
        CANCELED_BUYER, // 6
        CANCELED_SELLER, // 7 impossible state
        CLEARED_SELLER, // 8
        CLEARED_ARBITER // 9
    }
```

### Recommendation

It is recommended to review the contract's code and remove the `DealState` enum declarations that are not used in the contract's logic. Specifically, the `PAYMENT_COMPLETE`, `DISPUTE`, and `CANCELED_SELLER` states should be eliminated if they do not reflect any real functionality. Removing these redundant `enum` declarations will help in gas optimization by reducing the contract size and will also

enhance the clarity and maintainability of the contract by ensuring that only relevant states are represented in the code.

# L09 - Dead Code Elimination

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | DP2PCoin.sol#L42,66,91,101,120,134,153,163,180,190,207,247,272,288,324,358,403,416,431,447,482,501,515,528,725<br>DP2PToken.sol#L21,46,62,98,132,177,190,205,221,256,275,289,302,532,557,586,619,629,646,656,830,845,857<br>DP2PTokenLegacyUSDT.sol#L21,46,62,98,132,177,190,205,221,256,275,289,302,532,557,586,619,629,646,656 |
| **Status** | Unresolved |

## Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function isContract(address account) internal view returns
(bool) {
        // This method relies on
extcodesize/address.code.length, which returns 0
        // for contracts in construction, since the code is
only stored at the end
        // of the constructor execution.

        return account.code.length > 0;
    }

function sendValue(address payable recipient, uint256 amount)
internal {
        require(address(this).balance >= amount, "Address:
insufficient balance");

        (bool success, ) = recipient.call{value: amount}("");
        require(success, "Address: unable to send value,
recipient may have reverted");
    }

...
```

```
function toString(uint256 value) internal pure returns (string
memory) {
        // Inspired by OraclizeAPI's implementation - MIT licence
        //
https://github.com/oraclize/ethereum-api/blob/b42146b063c7d6ee13588
46c198246239e9360e8/oraclizeAPI_0.4.25.sol

        if (value == 0) {
            return "0";
...
        while (value != 0) {
            digits -= 1;
            buffer[digits] = bytes1(uint8(48 + uint256(value %
10)));
            value /= 10;
        }
        return string(buffer);
    }

...
```

```
function toString(uint256 value) internal pure returns (string
memory) {
        // Inspired by OraclizeAPI's implementation - MIT licence
        //
https://github.com/oraclize/ethereum-api/blob/b42146b063c7d6ee13588
46c198246239e9360e8/oraclizeAPI_0.4.25.sol

        if (value == 0) {
            return "0";
...
        while (value != 0) {
            digits -= 1;
            buffer[digits] = bytes1(uint8(48 + uint256(value %
10)));
            value /= 10;
        }
        return string(buffer);
    }

...
```

## Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

## L17 - Usage of Solidity Assembly

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | DP2PCoin.sol#L219,368<br>DP2PToken.sol#L142,685<br>DP2PTokenLegacyUSDT.sol#L142,685 |
| **Status** | Unresolved |

### Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata),
returndata_size)
                }

assembly {
                r := mload(add(signature, 0x20))
                s := mload(add(signature, 0x40))
                v := byte(0, mload(add(signature, 0x60)))
            }
```

```
assembly {
            ...
            }

assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata),
returndata_size)
                }
```

```
assembly {
            ...
             }

assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata),
returndata_size)
                }
```

## Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

## L18 - Multiple Pragma Directives

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | DP2PCoin.sol#L10,236,307,538,588,701<br>DP2PToken.sol#L10,81,312,362,476,702,788,900<br>DP2PTokenLegacyUSDT.sol#L10,81,312,362,476,702,787 |
| **Status** | Unresolved |

### Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

```
pragma solidity ^0.8.0;
pragma solidity ^0.8.0;
pragma solidity ^0.8.1;
```

### Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

## L19 - Stable Compiler Version

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | DP2PCoin.sol#L10,236,307,538,588,701<br>DP2PToken.sol#L10,81,312,362,476,702,788,900<br>DP2PTokenLegacyUSDT.sol#L10,81,312,362,476,702,787 |
| **Status** | Unresolved |

### Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.1;
pragma solidity ^0.8.0;
pragma solidity ^0.8.0;
```

### Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

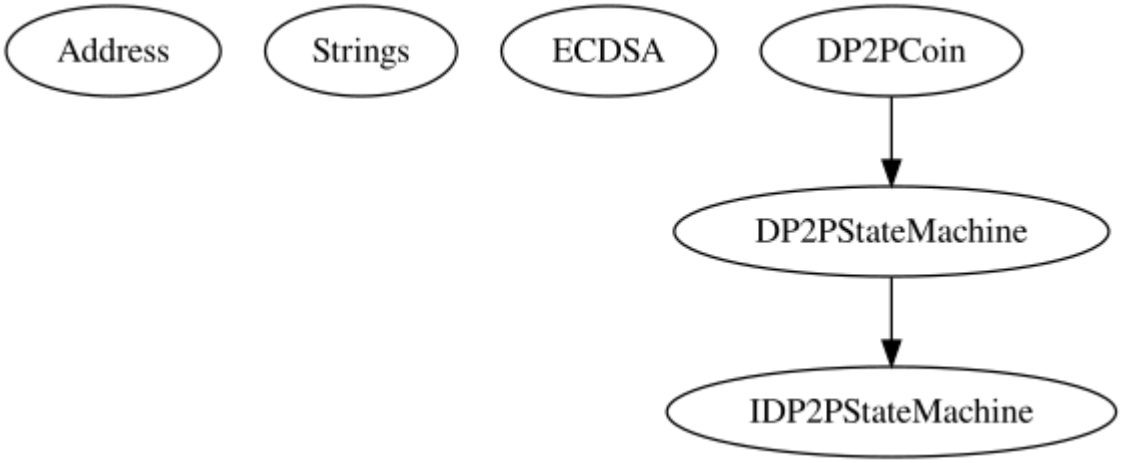
# Functions Analysis

| Contract | Type | Bases | | |
|---|---|---|---|---|
| | **Function Name** | **Visibility** | **Mutability** | **Modifiers** |
| | | | | |
| **Address** | Library | | | |
| | isContract | Internal | | |
| | sendValue | Internal | ✓ | |
| | functionCall | Internal | ✓ | |
| | functionCall | Internal | ✓ | |
| | functionCallWithValue | Internal | ✓ | |
| | functionCallWithValue | Internal | ✓ | |
| | functionStaticCall | Internal | | |
| | functionStaticCall | Internal | | |
| | functionDelegateCall | Internal | ✓ | |
| | functionDelegateCall | Internal | ✓ | |
| | verifyCallResult | Internal | | |
| | | | | |
| **Strings** | Library | | | |
| | toString | Internal | | |
| | toHexString | Internal | | |
| | toHexString | Internal | | |
| | | | | |

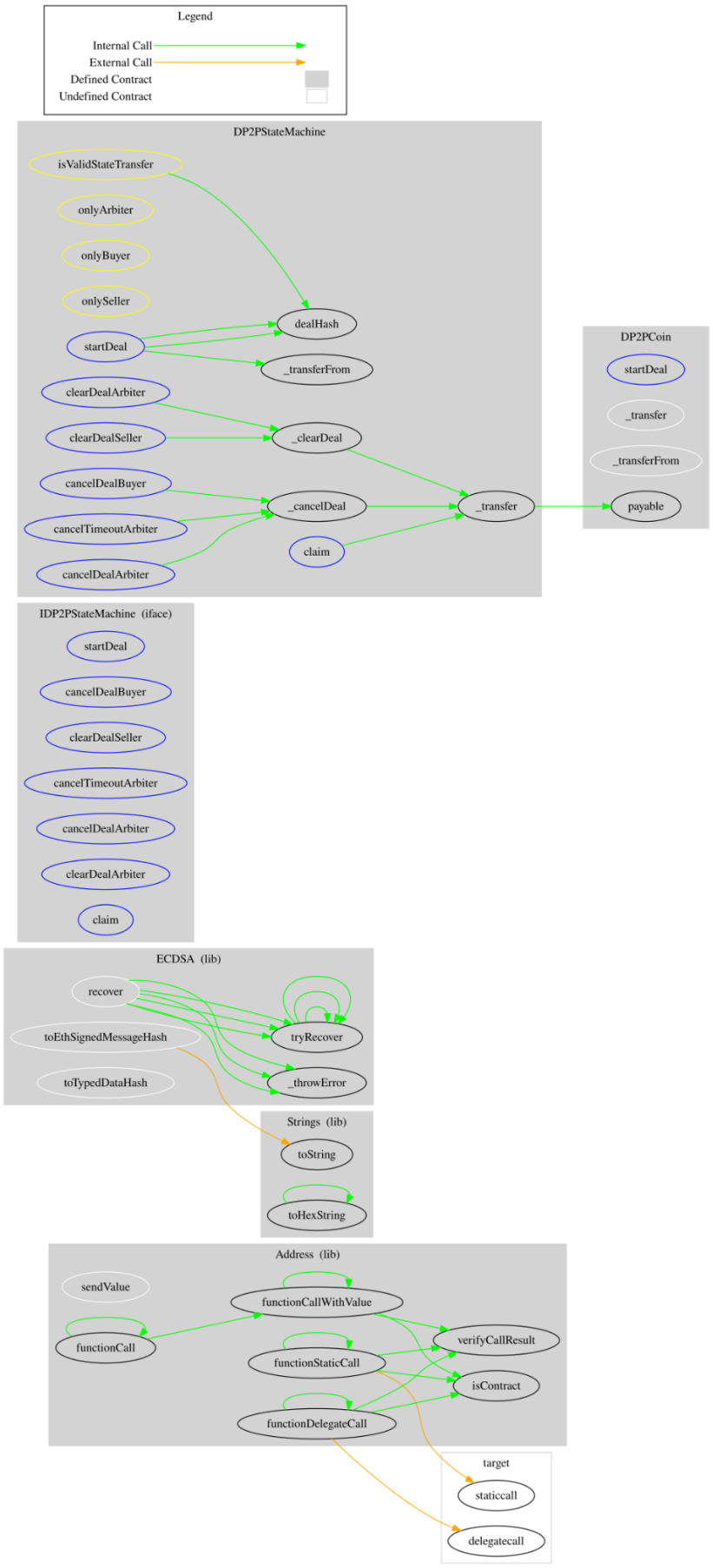
| | | | | |
|---|---|---|---|---|
| **ECDSA** | Library | | | |
| | _throwError | Private | | |
| | tryRecover | Internal | | |
| | recover | Internal | | |
| | tryRecover | Internal | | |
| | recover | Internal | | |
| | tryRecover | Internal | | |
| | recover | Internal | | |
| | toEthSignedMessageHash | Internal | | |
| | toEthSignedMessageHash | Internal | | |
| | toTypedDataHash | Internal | | |
| | | | | |
| **IDP2PStateMachine** | Interface | | | |
| | startDeal | External | Payable | - |
| | cancelDealBuyer | External | ✓ | - |
| | clearDealSeller | External | ✓ | - |
| | cancelTimeoutArbiter | External | ✓ | - |
| | cancelDealArbiter | External | ✓ | - |
| | clearDealArbiter | External | ✓ | - |
| | claim | External | ✓ | - |
| | | | | |
| **DP2PStateMachine** | Implementation | IDP2PState Machine | | |
| | startDeal | External | Payable | onlySeller |

| | | | | |
|---|---|---|---|---|
| | cancelDealBuyer | External | ✓ | onlyBuyer isValidStateTransfer |
| | clearDealSeller | External | ✓ | onlySeller isValidStateTransfer |
| | cancelTimeoutArbiter | External | ✓ | onlyArbiter isValidStateTransfer |
| | cancelDealArbiter | External | ✓ | onlyArbiter isValidStateTransfer |
| | clearDealArbiter | External | ✓ | onlyArbiter isValidStateTransfer |
| | claim | External | ✓ | - |
| | dealHash | Internal | | |
| | _cancelDeal | Internal | ✓ | |
| | _clearDeal | Internal | ✓ | |
| | _transfer | Internal | ✓ | |
| | _transferFrom | Internal | ✓ | |
| | | | | |
| **DP2PCoin** | Implementation | DP2PStateMachine | | |
| | startDeal | External | Payable | onlySeller |
| | _transfer | Internal | ✓ | |
| | _transferFrom | Internal | ✓ | |

# Inheritance Graph



# Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
DP2PStateMachine
isValidStateTransfer
onlyArbiter
onlyBuyer
onlySeller
startDeal
dealHash
_transferFrom
clearDealArbiter
clearDealSeller
_clearDeal
cancelDealBuyer
cancelTimeoutArbiter
cancelDealArbiter
_cancelDeal
claim
_transfer
DP2PCoin
startDeal
_transfer
_transferFrom
payable
IDP2PStateMachine (iface)
startDeal
cancelDealBuyer
clearDealSeller
cancelTimeoutArbiter
cancelDealArbiter
clearDealArbiter
claim
ECDSA (lib)
recover
toEthSignedMessageHash
toTypedDataHash
tryRecover
_throwError
Strings (lib)
toString
toHexString
Address (lib)
sendValue
functionCall
functionCallWithValue
functionStaticCall
functionDelegateCall
verifyCallResult
isContract
target
staticcall
delegatecall

# Summary

The SMARTSWAP contract suite enables decentralized peer-to-peer transactions across multiple blockchains, with specific functionalities for native cryptocurrencies, ERC20 tokens, and the USDT on TRON. It provides mechanisms for initiating, managing, canceling, and concluding deals, along with the allocation and claiming of transaction fees, ensuring a secure and transparent trading environment. There is no outside influence on contracts.

# Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment 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.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

# About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

**The Cyberscope team**

https://www.cyberscope.io