Cyberscope

Audit Report

ShadowGold

April 2024

SHA256 fc503bfe3c1c8c2d782f5f2f6def8f656f595fbce746f11e066c0690efcfe59a

SHA256 4ebfe27c579c115fbb09330b1e75395671400ee39acc3f57afb4c261f82104ab

Audited by © cyberscope

Table of Contents

Review

SDG.sol	https://testnet.bscscan.com/address/0x4f042268E4EDd7541C71a1035644FC50648318BC
DividendDistributor.sol	https://testnet.bscscan.com/address/0x377fA8401b2105eb9C2Ffe1cE68A6D175C5f7112

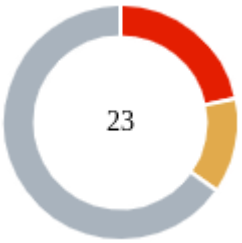
Audit Updates

Initial Audit	03 Apr 2024 https://github.com/cyberscope-io/audits/blob/main/shadowfi/v1/audit.pdf
Corrected Phase 2	15 Apr 2024

Source Files

Filename	SHA256
SDGDistributor.sol	fc503bfe3c1c8c2d782f5f2f6def8f656f595fbce746f11e066c0690efcfe59a
SDG.sol	4ebfe27c579c115fbb09330b1e75395671400ee39acc3f57afb4c261f82104ab

Findings Breakdown



Critical	5
Medium	3
Minor / Informative	15

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	5	0	0	0
Medium	2	1	0	0
Minor / Informative	15	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	ELFM	Exceeds Fees Limit	Unresolved
●	MT	Mints Tokens	Unresolved
●	ST	Stops Transactions	Unresolved
●	US	Untrusted Source	Unresolved
●	ZD	Zero Division	Unresolved
●	BC	Blacklists Addresses	Unresolved
●	IMC	Ineffective Minting Check	Unresolved
●	UDB	Unupdated Distributor Balances	Acknowledged
●	DDP	Decimal Division Precision	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	MMN	Misleading Method Naming	Unresolved
●	MIV	Missing Index Verification	Unresolved
●	PAV	Pair Address Validation	Unresolved
●	PLPI	Potential Liquidity Provision Inadequacy	Unresolved

●	PVC	Price Volatility Concern	Unresolved
●	RFV	Redundant Fee Variable	Unresolved
●	RRS	Redundant Require Statement	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved
●	OCTD	Transfers Contract's Tokens	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

ELFM - Exceeds Fees Limit

Criticality	Critical
Location	DividendDistributor.sol#L756
Status	Unresolved

Description

The contract owner has the authority to increase the fees over the allowed limit of 25%. The owner may take advantage of it by calling the `setLaunchedAt` function of the token contract with any arbitrary number effectively modifying the result of the `checkLaunched` in order to return a high percentage fee value.

```
if (ISDG(_token).checkLaunched() + 1 >= block.number) {
    return feeDenominator.sub(1);
}
```

Recommendation

The contract could embody a check for the maximum acceptable value. The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

MT - Mints Tokens

Criticality	Critical
Location	SDG.sol#L714
Status	Unresolved

Description

The contract allows the owner to change and designate any address as the distributor via the `setDistributorAndFeeReceiverContract` function. This grants the owner unchecked authority to manipulate the distribution mechanism, by calling the `processFee` function to mint tokens to the `distributor` address. As a result, the contract tokens will be highly inflated.

```
    function processFee(address sender, uint256 feeAmount) external {
        require(
            msg.sender == address(distributor),
            "Only Distributor can Process Fee."
        );
        _balances[address(distributor)] =
_balances[address(distributor)].add(
            feeAmount
        );
        emit Transfer(sender, address(distributor), feeAmount);
    }
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.

- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

ST - Stops Transactions

Criticality	Critical
Location	SDG.sol#L630
Status	Unresolved

Description

The transactions are initially disabled for all users excluding the authorized addresses. The owner can enable the transactions for all users. Once the transactions are enable the owner will not be able to disable them again.

```
if (!allowedAddresses[msg.sender] &&
!allowedAddresses[recipient]) {
    require(
        block.timestamp > transferBlockTime,
        "Transfers have not been enabled yet."
    );
```

Additionally, the contract owner has the authority to stop transactions, as described in detail in sections `US` , and `ZD` . As a result, the contract might operate as a honeypot.

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions. Some suggestions are:

- Introduce a multi-sign wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

US - Untrusted Source

Criticality	Critical
Location	SDG.sol#L791
Status	Unresolved

Description

The contract uses an external contract in order to determine the transaction's flow. The external contract is untrusted. As a result, it may produce security issues and harm the transactions.

```
function setDistributorAndFeeReceiverContract(
    address _distributorContract,
    address _feeReceiver
) public onlyOwner {
    distributor = IDividendDistributor(_distributorContract);
    isDividendExempt[_distributorContract] = true;
    isFeeExempt[_feeReceiver] = true;
}
```

Recommendation

The contract should use a trusted external source. A trusted source could be either a commonly recognized or an audited contract. The pointing addresses should not be able to change after the initialization.

ZD - Zero Division

Criticality	Critical
Location	DividendDistributor.sol#L774,789,907
Status	Unresolved

Description

The contract is using variables that may be set to zero as denominators. This can lead to unpredictable and potentially harmful results, such as a transaction revert.

Specifically the `feeDenominator` , `buybackMultiplierDenominator` and `totalBuyFee` variables can be set to zero.

```
        uint256 feeAmount =
amount.mul(getTotalFee(isSell(recipient))).div(
            feeDenominator
         ...
         uint256 feeIncrease = totalFee
             .mul(buybackMultiplierNumerator)
             .div(buybackMultiplierDenominator)
             .sub(totalFee);
         ...
         uint256 swapAmount =
swapThreshold.mul(marketingFee).div(totalBuyFee);
```

Recommendation

It is important to handle division by zero appropriately in the code to avoid unintended behavior and to ensure the reliability and safety of the contract. The contract should ensure that the divisor is always non-zero before performing a division operation. It should prevent the variables to be set to zero, or should not allow the execution of the corresponding statements.

BC - Blacklists Addresses

Criticality	Medium
Location	SDG.sol#L636
Status	Unresolved

Description

The contract owner has the authority to stop addresses from transactions. The owner may take advantage of it by calling the `setBlackListed` function.

```
require(
    !blackList[sender] && !blackList[recipient],
    "Either the spender or recipient is blacklisted."
);
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

IMC - Ineffective Minting Check

Criticality	Medium
Location	SDG.sol#L724
Status	Unresolved

Description

The contract includes the `processFee` function that is designed to prevents the minting of new tokens by utilizing require statements. Specifically, it checks to ensure that the `_totalSupply` equals `currentSupply` after an attempt to update balances, implying that no new tokens should be minted during the transaction. However, the functionality does not work as intended because `currentSupply` is set to `_totalSupply` before any operations that could increase `_totalSupply` are performed and it is not updated afterwards. This results in the comparison being redundant and ineffective as `currentSupply` and `_totalSupply` are always equal at the point of comparison, thereby never actually preventing minting.

```
function processFee(address sender, uint256 feeAmount) external
{
    require(
        msg.sender == address(distributor),
        "Only Distributor can Process Fee."
    );
    uint256 currentSupply = _totalSupply;
    _balances[address(distributor)] =
_balances[address(distributor)].add(
        feeAmount
    );
    require(
        _totalSupply == currentSupply,
        "Minting new tokens is not allowed."
    );
    emit Transfer(sender, address(distributor), feeAmount);
}
```

Recommendation

It is recommended to reconsider the check that should be in place to effectively prevent unauthorized minting. If the intention is to cap the token supply, the contract should enforce

an upper limit for `_totalSupply`. This can be achieved by setting a predefined maximum supply limit (maxSupply) and updating the require condition to check whether `_totalSupply` exceeds this maxSupply after any token generation operations. This would ensure that no additional tokens beyond the set limit can be minted, aligning with standard practices for fixed-supply token ecosystems.

UDB - Unupdated Distributor Balances

Criticality	Medium
Location	SDG.sol#L640
Status	Acknowledged

Description

The contract contains the `_transferFrom` which immediately executes the `_basicTransfer` function in case the `if` condition condition is met, resulting in a transfer transaction before updating the balances of the distributor. Consequently, the updated balances will not be reflected accurately in the distribution contract, leading to discrepancies in token distribution.

```
    function _transferFrom(
         address sender,
         address recipient,
         uint256 amount
     ) internal returns (bool) {
     ...

     if (IDividendDistributor(distributor).checkInSwap()) {
          return _basicTransfer(sender, recipient, amount);
      }
     ...
```

Recommendation

It is recommended to ensure that balances of the distributor are updated before executing any transfer transactions. Implement mechanisms to update the distributor balances synchronously with token transfers to maintain consistency in token distribution.

Team Update

The team has acknowledged that this is not a security issue and states:

Distributor itself never has shares. It is exempted. The only time checkInSwap is true, is if the distributor is swapping to the LP address. The LP address is also exempted from shares.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	DividendDistributor.sol#L919
Status	Unresolved

Description

Division of decimal (fixed point) numbers can result in rounding errors due to the way that division is implemented in Solidity. Thus, it may produce issues with precise calculations with decimal numbers.

Solidity represents decimal numbers as integers, with the decimal point implied by the number of decimal places specified in the type (e.g. decimal with 18 decimal places). When a division is performed with decimal numbers, the result is also represented as an integer, with the decimal point implied by the number of decimal places in the type. This can lead to rounding errors, as the result may not be able to be accurately represented as an integer with the specified number of decimal places.

Hence, the splitted shares will not have the exact precision and some funds may not be calculated as expected.

```
uint256 swapAmount =
swapThreshold.mul(marketingFee).div(totalBuyFee);
...
uint256 amountSDGReflection =
swapThreshold.mul(reflectionFee).div(
    totalBuyFee
);
uint256 amountSDGReceiver =
swapThreshold.mul(sdgReceiverFee).div(
    totalBuyFee
);
uint256 amountSDGBuyback = swapThreshold.mul(buybackFee).div(
    totalBuyFee
```

Recommendation

The team is advised to take into consideration the rounding results that are produced from the solidity calculations. The contract could calculate the subtraction of the divided funds in the last calculation in order to avoid the division rounding issue.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	SDG.sol#L393,394
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
maticPair
paxgPair
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

MMN - Misleading Method Naming

Criticality	Minor / Informative
Location	SDG.sol#L856
Status	Unresolved

Description

Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand. Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand.

Specifically, the `getCirculatingSupply` function calculate the circulating supply but behaves similarly to `getMaxCirculatingSupply`, since this function subtracts the balance of a "DEAD" address and a "ZERO" address from the maximum supply, potentially providing misleading information about the actual circulating supply.

```
   function getCirculatingSupply() public view returns
(uint256) {
        return
_maxSupply.sub(balanceOf(DEAD)).sub(balanceOf(ZERO));
    }
```

Recommendation

It's always a good practice for the contract to contain method names that are specific and descriptive. The team is advised to keep in mind the readability of the code. It is recommended to revise the `getCirculatingSupply` function to accurately reflect the circulating supply by excluding addresses that are not actively participating in the circulation from the total supply.

MIV - Missing Index Verification

Criticality	Minor / Informative
Location	SDG.sol#L272,286,300
Status	Unresolved

Description

The contract contains the `authorizeForMultiplePermissions`, `unauthorizeFor`, and `unauthorizeForMultiplePermissions` functions, all relying on `permIndex` for permission handling. However, these functions lack verification to ensure the existence of the `permIndex` before its usage. Consequently, users could define a `permIndex` that does not exist, leading to potential unauthorized access or unintended behavior.

```
    function authorizeForMultiplePermissions(
        address adr,
        string[] calldata permissionNames
    ) public authorizedFor(Permission.Authorize) {
        for (uint256 i; i < permissionNames.length; i++) {
            uint256 permIndex =
permissionNameToIndex[permissionNames[i]];
            authorizations[adr][permIndex] = true;
            emit AuthorizedFor(adr, permissionNames[i],
permIndex);
        }
    }

    function unauthorizeFor(
        address adr,
        string memory permissionName
    ) public authorizedFor(Permission.Unauthorize) {
        require(adr != owner, "Can't unauthorize owner");

        uint256 permIndex =
permissionNameToIndex[permissionName];
        authorizations[adr][permIndex] = false;
        emit UnauthorizedFor(adr, permissionName, permIndex);
    }

    function unauthorizeForMultiplePermissions(
        address adr,
        string[] calldata permissionNames
    ) public authorizedFor(Permission.Unauthorize) {
        require(adr != owner, "Can't unauthorize owner");

        for (uint256 i; i < permissionNames.length; i++) {
            uint256 permIndex =
permissionNameToIndex[permissionNames[i]];
            authorizations[adr][permIndex] = false;
            emit UnauthorizedFor(adr, permissionNames[i],
permIndex);
        }
    }
```

Recommendation

It is recommended to enhance the functions by incorporating additional checks to validate the existence of the `permIndex` before executing operations. This would mitigate the risk of unauthorized access and ensure the contract behaves as intended.

PAV - Pair Address Validation

Criticality	Minor / Informative
Location	DividendDistributor.sol#L633,819,843
Status	Unresolved

Description

The contract is missing address validation in the pair address argument. The absence of validation reveals a potential vulnerability, as it lacks proper checks to ensure the integrity and validity of the pair address provided as an argument. The pair address is a parameter used in certain methods of decentralized exchanges for functions like token swaps and liquidity provisions.

The absence of address validation in the pair address argument can introduce security risks and potential attacks. Without proper validation, if the owner's address is compromised, the contract may lead to unexpected behavior like loss of funds.

```
address[] memory path = new address[](2);
path[0] = address(_token);
path[1] = address(PAXG);
...
function buyTokensWETH(uint256 amount, address to) internal
swapping {
    address[] memory path = new address[](2);
    path[0] = address(WETH);
    path[1] = address(_token);
    ...

function buyTokensPAXG(uint256 amount, address to) internal
swapping {
    address[] memory path = new address[](2);
    path[0] = address(PAXG);
    path[1] = address(_token);
```

Recommendation

To mitigate the risks associated with the absence of address validation in the pair address argument, it is recommended to implement comprehensive address validation mechanisms. A recommended approach could be to verify pair existence in the decentralized application. Prior to interacting with the pair address contract, perform checks to verify the existence and validity of the contract at the provided address. This can be achieved by querying the provider's contract or utilizing external libraries that provide contract verification services.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	DividendDistributor.sol#L648
Status	Unresolved

Description

The contract operates under the assumption that liquidity is consistently provided to the pair between the contract's token and the native currency. However, there is a possibility that liquidity is provided to a different pair. This inadequacy in liquidity provision in the main pair could expose the contract to risks. Specifically, during eligible transactions, where the contract attempts to swap tokens with the main pair, a failure may occur if liquidity has been added to a pair other than the primary one. Consequently, transactions triggering the swap functionality will result in a revert.

```
address[] memory path = new address[](2);
path[0] = address(_token);
path[1] = address(PAXG)
router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
    amount,
    0,
    path,
    address(this),
    block.timestamp
);
```

Recommendation

The team is advised to implement a runtime mechanism to check if the pair has adequate liquidity provisions. This feature allows the contract to omit token swaps if the pair does not have adequate liquidity provisions, significantly minimizing the risk of potential failures.

Furthermore, the team could ensure the contract has the capability to switch its active pair in case liquidity is added to another pair.

Additionally, the contract could be designed to tolerate potential reverts from the swap functionality, especially when it is a part of the main transfer flow. This can be achieved by

executing the contract's token swaps in a non-reversible manner, thereby ensuring a more resilient and predictable operation.

PVC - Price Volatility Concern

Criticality	Minor / Informative
Location	DividendDistributor.sol#L907
Status	Unresolved

Description

The contract accumulates tokens from the taxes to swap them for ETH. The variable `swapThreshold` sets a threshold where the contract will trigger the swap functionality. If the variable is set to a big number, then the contract will swap a huge amount of tokens for ETH.

It is important to note that the price of the token representing it, can be highly volatile. This means that the value of a price volatility swap involving Ether could fluctuate significantly at the triggered point, potentially leading to significant price volatility for the parties involved.

```
    function swapBack() public swapping onlyToken {
        uint256 swapAmount =
swapThreshold.mul(marketingFee).div(totalBuyFee);
```

Recommendation

The contract could ensure that it will not sell more than a reasonable amount of tokens in a single transaction. A suggested implementation could check that the maximum amount should be less than a fixed percentage of the exchange reserves. Hence, the contract will guarantee that it cannot accumulate a huge amount of tokens in order to sell them.

RFV - Redundant Fee Variable

Criticality	Minor / Informative
Location	DividendDistributor.sol#L922,942
Status	Unresolved

Description

The contract is utilizing two separate variables, `sdgReceiverFee` and `buybackFee`, to calculate the total amount to be transferred to the `sdgFeeReceiver`. These variables are used to determine portions of the `swapThreshold` that correspond to different fees before being summed up for the final transfer amount. The calculation involves multiplying the `swapThreshold` by each fee and dividing by the `totalBuyFee`, resulting in `amountSDGReceiver` and `amountSDGBuyback` respectively. Subsequently, these amounts are added together for the transfer to sdgFeeReceiver. This approach, while mathematically sound, introduces unnecessary complexity and redundancy since the addition of these variables does not implement any additional functionality or differentiation in the handling of fees. Essentially, the contract is performing an extra step without a clear benefit, which could lead to confusion, increased gas costs, and potential errors in future modifications.

```
uint256 amountSDGReceiver =
swapThreshold.mul(sdgReceiverFee).div(
    totalBuyFee
);
uint256 amountSDGBuyback = swapThreshold.mul(buybackFee).div(
    totalBuyFee
);
...
if (amountSDGReceiver > 0 || amountSDGBuyback > 0) {
    try
        IERC20(address(_token)).transfer(
            sdgFeeReceiver,
            amountSDGReceiver + amountSDGBuyback
        )
    {
        emit ReceiverAmount(amountSDGBuyback,
amountSDGReceiver);
```

Recommendation

It is recommended to simplify the fee structure by consolidating `sdgReceiverFee` and `buybackFee` into a single variable. This can be achieved by either combining their values into a single fee variable or by re-evaluating the necessity of distinguishing these fees if they ultimately serve a similar purpose and are directed to the same receiver. Simplification will not only reduce the contract's complexity but also minimize potential points of failure and optimize gas costs associated with these calculations and transactions. Additionally, this change would make the contract more straightforward, enhancing its readability and maintainability. Future updates or audits will benefit from a clearer understanding of the fee handling mechanism, thereby reducing the risk of errors or unintended consequences.

RRS - Redundant Require Statement

Criticality	Minor / Informative
Location	SDG.sol#L64 DividendDistributor.sol#L64
Status	Unresolved

Description

The contract utilizes a `require` statement within the `add` function aiming to prevent overflow errors. This function is designed based on the SafeMath library's principles. In Solidity version 0.8.0 and later, arithmetic operations revert on overflow and underflow, making the overflow check within the function redundant. This redundancy could lead to extra gas costs and increased complexity without providing additional security.

```
function add(uint256 a, uint256 b) internal pure returns
(uint256) {
      uint256 c = a + b;
      require(c >= a, "SafeMath: addition overflow");
      return c;
}
```

Recommendation

It is recommended to remove the `require` statement from the add function since the contract is using a Solidity pragma version equal to or greater than 0.8.0. By doing so, the contract will leverage the built-in overflow and underflow checks provided by the Solidity language itself, simplifying the code and reducing gas consumption. This change will uphold the contract's integrity in handling arithmetic operations while optimizing for efficiency and cost-effectiveness.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	SDG.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily in cases where the explanatory error message is not used.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library in cases where the revert error message is not used. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/v0.8.16/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	SDG.sol#L923 DividendDistributor.sol#L969
Status	Unresolved

Description

The contract owner has the authority to claim all the balance of the contract. The owner may take advantage of it by calling the `withdrawTokens` function.

```
   function withdrawTokens(address _token, uint256 _amount)
public onlyOwner {
        IERC20(_token).transfer(owner, _amount);
    }
...
    function withdrawTokens(address token, uint256 _amount)
public onlyOwner {
        IERC20(token).transfer(owner, _amount);
    }
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	SDG.sol#L352,353,354,356,357,358,361,362,560,597,629,640,646,655,667,671,692,700,705,706
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
string constant _name = "Test SDG"
string constant _symbol = "TSDG"
uint8 constant _decimals = 9
uint256 constant _totalSupply = 10 ** 8 * (10 ** _decimals)
uint256 constant _maxSupply = 10 ** 8 * (10 ** _decimals)
uint256 public _maxTxAmount
mapping (address => uint256) _balances
mapping (address => mapping (address => uint256)) _allowances
address _feeReceiver
address _distributorContract
uint256 _minPeriod
uint256 _minDistribution
uint256 GWEI
uint256 _amount

...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	SDG.sol#L529,618,673
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
totalShares =
totalShares.sub(shares[shareholder].amount).add(amount)
launchedAt = launched_
transferBlockTime += _addSeconds
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	SDG.sol#L156,205,229,264
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
uint256 i
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	SDG.sol#L668
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
IERC20(_token).transfer(owner, _amount)
```

Recommendation

The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
SafeMath	Library			
	add	Internal		
	sub	Internal		
	sub	Internal		
	mul	Internal		
	div	Internal		
	div	Internal		
IERC20	Interface			
	totalSupply	External		-
	decimals	External		-
	symbol	External		-
	name	External		-
	getOwner	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-

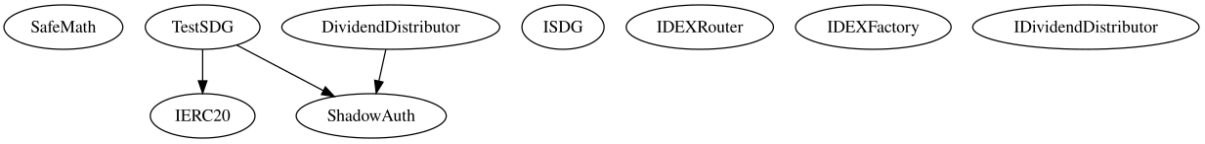
IDEXFactory	Interface			
	createPair	External	✓	-
IDEXRouter	Interface			
	factory	External		-
ShadowAuth	Implementation			
		Public	✓	-
	authorizeFor	Public	✓	authorizedFor
	authorizeForMultiplePermissions	Public	✓	authorizedFor
	unauthorizeFor	Public	✓	authorizedFor
	unauthorizeForMultiplePermissions	Public	✓	authorizedFor
	isOwner	Public		-
	isAuthorizedFor	Public		-
	isAuthorizedFor	Public		-
	transferOwnership	Public	✓	onlyOwner
	getPermissionNameToIndex	Public		-
	getPermissionUnlockTime	Public		-
	isLocked	Public		-
	lockPermission	Public	✓	authorizedFor
	unlockPermission	Public	✓	-

IDividendDistributor	Interface			
	setDistributionCriteria	External	✓	-
	setShare	External	✓	-
	deposit	External	✓	-
	process	External	✓	-
	claimDividend	External	✓	-
	takeFee	External	✓	-
	shouldTakeFee	External		-
	swapBack	External	✓	-
	shouldSwapBack	External		-
	isPair	External		-
	checkInSwap	External		-
TestSDG	Implementation	IERC20, ShadowAuth		
		Public	✓	ShadowAuth
		External	Payable	-
	totalSupply	External		-
	decimals	External		-
	symbol	External		-
	name	External		-
	getOwner	External		-
	balanceOf	Public		-
	allowance	External		-

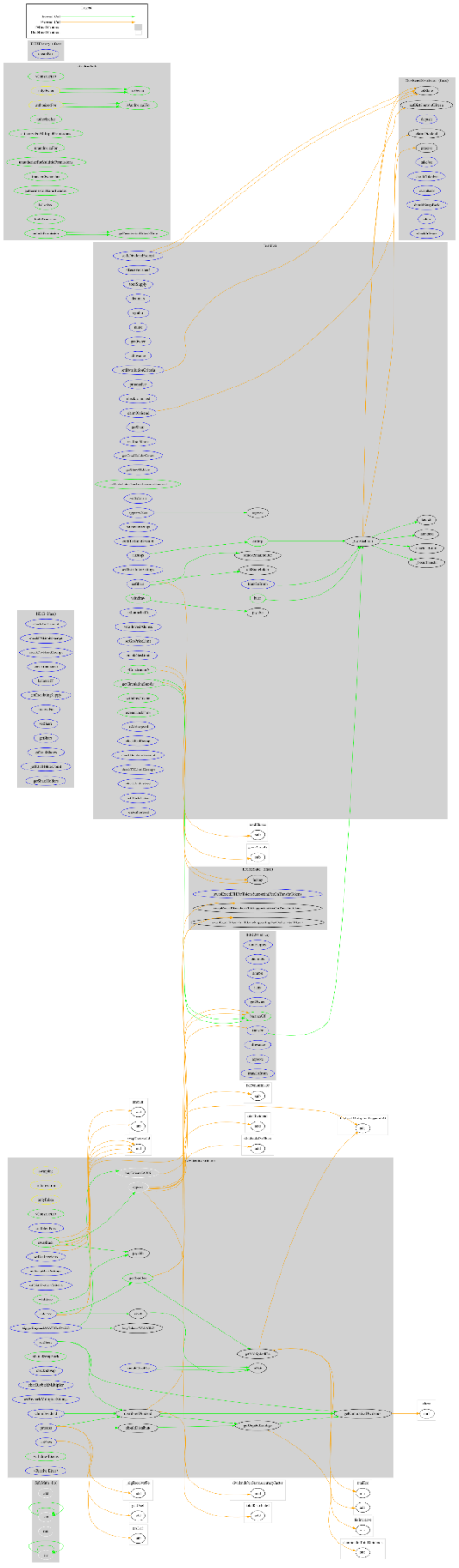
	approve	Public	✓	-
	approveMax	External	✓	-
	transfer	External	✓	-
	transferFrom	External	✓	-
	_transferFrom	Internal	✓	
	_basicTransfer	Internal	✓	
	checkTxLimit	Internal		
	processFee	External	✓	-
	checkLaunched	External		-
	launched	Internal		
	launch	Internal	✓	
	setShare	External	✓	-
	getShare	External		-
	getTotalShares	External		-
	getTotalHolderCount	External		-
	getShareHolders	External		-
	addShareholder	Internal	✓	
	removeShareholder	Internal	✓	
	setDistributorAndFeeReceiverContract	Public	✓	onlyOwner
	setTxLimit	External	✓	authorizedFor
	setIsDividendExempt	External	✓	-
	setIsFeeExempt	External	✓	-
	setIsTxLimitExempt	External	✓	-

	setDistributionCriteria	External	✓	authorizedFor
	setDistributorSettings	External	✓	authorizedFor
	getCirculatingSupply	Public		-
	claimDividend	External	✓	-
	setLaunchedAt	External	✓	authorizedFor
	setAllowedAddress	External	✓	onlyOwner
	setGasPriceLimit	External	✓	onlyOwner
	enableGasLimit	External	✓	onlyOwner
	burn	Public	✓	-
	airdrop	Public	✓	onlyOwner
	airdrops	External	✓	onlyOwner
	withdraw	Public	✓	onlyOwner
	withdrawTokens	Public	✓	onlyOwner
	extendLockTime	Public	✓	onlyOwner
	isAirdropped	External		-
	checkFeeExempt	External		-
	checkDividendExempt	External		-
	checkTXLimitExempt	External		-
	checkAuthorized	External		-
	setBlackListed	External	✓	onlyOwner
	setAuthorized	External	✓	onlyOwner

Inheritance Graph



Flow Graph



Summary

ShadowGold contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. There are some functions that can be abused by the owner like stop transactions, manipulate the fees and blacklist addresses. A multi-wallet signing pattern will provide security against potential hacks. Temporarily locking the contract or renouncing ownership will eliminate all the contract threats.

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