Cyberscope

Audit Report

ShadowFi

April 2024

SHA256 277b9567698f3abe4eae79e62cb45de10f89b2d6bbddab1126b1f4e4f49f4a1f

Audited by © cyberscope

Table of Contents

Table of Contents **1**
Review **3**
Audit Updates 3
Source Files 3
Overview **4**
Liquidity Locking and Management 4
Liquidity Withdrawal 4
Liquidity Migration 4
Token Injection and Burning 5
View Functions 5
Findings Breakdown **6**
Diagnostics **7**
CR - Code Repetition 9
Description 9
Recommendation 9
CCR - Contract Centralization Risk 10
Description 10
Recommendation 11
LRV - Liquidity Removal Volatility 12
Description 12
Recommendation 13
LDP - Loss Division Precision 14
Description 14
Recommendation 14
MEN - Misleading Event Naming 16
Description 16
Recommendation 16
MC - Missing Check 18
Description 18
Recommendation 18
MU - Modifiers Usage 19
Description 19
Recommendation 19
PLPI - Potential Liquidity Provision Inadequacy 20
Description 20
Recommendation 20
RED - Redudant Event Declaration 22
Description 22
Recommendation 22

TSI - Tokens Sufficiency Insurance 23
Description 23
Recommendation 23
UVF - Unutilized View Functions 25
Description 25
Recommendation 26
L02 - State Variables could be Declared Constant 28
Description 28
Recommendation 28
L04 - Conformance to Solidity Naming Conventions 29
Description 29
Recommendation 29
L13 - Divide before Multiply Operation 31
Description 31
Recommendation 31
L14 - Uninitialized Variables in Local Scope 32
Description 32
Recommendation 32
L18 - Multiple Pragma Directives 33
Description 33
Recommendation 33
L19 - Stable Compiler Version 34
Description 34
Recommendation 34
L20 - Succeeded Transfer Check 35
Description 35
Recommendation 35
Functions Analysis 36
Inheritance Graph 38
Flow Graph 39
Summary 40
Disclaimer 41
About Cyberscope 42

Review

Testing Deploy	https://testnet.bscscan.com/address/0x323fd73018fe0dc40e40661bd9874d3a4149c477

Audit Updates

Initial Audit	03 Apr 2024

Source Files

Filename	SHA256
contracts/ShadowGoldLiquidityLocker.sol	277b9567698f3abe4eae79e62cb45de10f89b2d6bbddab1126b1f4e4f49f4a1f
contracts/ISwapRouter.sol	3ed2103ea82aa3ff86c0e7bc5687c06b54a4e1f381c7fdb9523aa0e25bcb6e26
@uniswap/v3-core/contracts/interfaces/callback/IUniswapV3SwapCallback.sol	171a9a692e71b6d532df655695b0b672bd8ea5dcca3b3363131700b45b0171c6

Overview

The `ShadowFiLiquidityLock` contract is designed to manage liquidity for the ShadowGoldToken within a decentralized finance (DeFi) ecosystem, specifically focusing on liquidity locking, migration, and management functionalities. It interacts with decentralized exchange (DEX) routers for swapping tokens and managing liquidity pairs.

Liquidity Locking and Management

This functionality is foundational to the contract's purpose, focusing on locking liquidity tokens to ensure stability and trust in the token's market. The contract initializes with a set lock time, during which liquidity tokens cannot be moved. Owner can extend this lock time, ensuring that liquidity is locked for the desired duration. The `endLock` function allows the contract owner to transfer locked liquidity tokens back to their address once the lock period has ended, signifying the end of the lock.

Liquidity Withdrawal

The contract provides functionalities for the contract owner to withdraw Ether or any ERC-20 tokens from the contract, excluding the liquidity provider (LP) tokens that are locked. This allows the recovery of any tokens accidentally sent to the contract or the withdrawal of tokens that are not subject to the lock conditions. It ensures that the contract can manage its funds effectively while adhering to the lock-up terms for liquidity tokens.

Liquidity Migration

The contract includes features to migrate liquidity from one pair to another or to change the composition of the liquidity provided. This is achieved through functions that allow the removal of liquidity from one pair and the addition to another, effectively enabling the contract to respond to changes in market conditions or strategic shifts in liquidity provision. This functionality is essential for maintaining optimal liquidity distribution across different trading pairs.

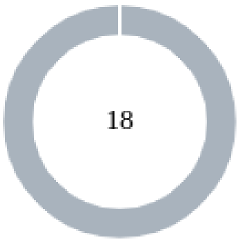
Token Injection and Burning

The contract encompasses functionalities for both active token supply management and liquidity enhancement, primarily through the `shadowBurst` function for token burning and the `injectWeth` and `injectPaxg` functions for liquidity injections. By removing liquidity, swapping tokens, and burning a portion of `ShadowGoldTokens`, the contract dynamically adjusts the token's supply, increasing its scarcity and value. Concurrently, direct liquidity injections of WETH or PAXG into their respective pairs aim to improve the market depth and stability. These mechanisms collectively serve to actively influence the token's economic environment, balancing supply adjustments with liquidity enhancements to respond to market demands and conditions effectively.

View Functions

The contract provides several view functions that allow anyone to query the current state of liquidity, such as the percentage of total liquidity owned by the contract, the amount of liquid tokens, and the lock time. These functions are crucial for transparency, allowing users and interested parties to verify the contract's actions and the status of the liquidity it controls.

Findings Breakdown



- Critical 0
- Medium 0
- Minor / Informative 18

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	18	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	CR	Code Repetition	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	LRV	Liquidity Removal Volatility	Unresolved
●	LDP	Loss Division Precision	Unresolved
●	MEN	Misleading Event Naming	Unresolved
●	MC	Missing Check	Unresolved
●	MU	Modifiers Usage	Unresolved
●	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
●	RED	Redudant Event Declaration	Unresolved
●	TSI	Tokens Sufficiency Insurance	Unresolved
●	UVF	Unutilized View Functions	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L13	Divide before Multiply Operation	Unresolved

●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L18	Multiple Pragma Directives	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

CR - Code Repetition

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#L368
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

Specifically the functions `shadowBurst` , `migrateSDG` and `migrateLP` use similar code segments.

```
function shadowBurst(uint256 percent, bool wethOrPaxg) public
onlyOwner {
    ...
}

function migrateSDG(uint256 percent, bool wethOrPaxg) public
onlyOwner {
    ...
}

function migrateLP(uint256 percent, bool wethOrPaxg) public
onlyOwner {
    ...
}
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#L333
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

Specifically the smart contract presents a centralization risk by granting the owner extensive control over locked tokens and the ability to extend lock periods. This centralized authority allows the owner to claim all tokens after the lock period and adjust the lock duration at will, posing trust and security concerns.

```
    function endLock() public onlyOwner {
        require(block.timestamp >= lockTime, "LP tokens are still
locked.");

        wethPair.transfer(owner(), wethPair.balanceOf(address(this)));
        paxgPair.transfer(owner(), wethPair.balanceOf(address(this)));

        lockEnded = true;
        emit LockEnded();
    }

    function extendLockTime(uint256 _extraLockTime) public onlyOwner {
        require(!lockEnded, "You already claimed all LP tokens.");
        require(_extraLockTime > 0, "Invalid extra lock time is
provided.");

        lockTime += _extraLockTime;
        emit ParameterUpdated();
    }

    function withdraw() public onlyOwner {
        uint256 balance = address(this).balance;
        payable(msg.sender).transfer(balance);
    }
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

LRV - Liquidity Removal Volatility

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#L399,446,497,545,598,664
Status	Unresolved

Description

The contract is designed to facilitate the removal of liquidity from token pairs, such as `ShadowGoldToken` with `WETH` and `PAXG`. This functionality, while providing flexibility in managing liquidity, poses a significant risk if a large portion of the liquidity is withdrawn in a single transaction. Such substantial liquidity removals can lead to increased price volatility of the involved tokens, potentially destabilizing the market and affecting investor confidence. The inherent risk is exacerbated by the lack of safeguards against the withdrawal of large liquidity percentages, which could lead to scenarios where the token's price becomes highly volatile, affecting all market participants.

```
(uint256 amountToken, uint256 amountWeth) =
router.removeLiquidity(
    address(shadowGoldToken),
    address(weth),
    removeAmount,
    0,
    0,
    address(this),
    block.timestamp + 120

(uint256 amountToken, uint256 amountPaxg) =
router.removeLiquidity(
    address(shadowGoldToken),
    address(paxg),
    removeAmount,
    0,
    0,
    address(this),
    block.timestamp + 120
);
...
(uint256 amountToken, uint256 amountWeth) =
router.removeLiquidity(
    address(shadowGoldToken),
    address(weth),
    removeAmount,
    0,
    0,
    address(this),
    block.timestamp + 120
);
...
```

Recommendation

It is recommended to implement safeguards within the contract to prevent the removal of large amounts of liquidity in a single operation. This could involve setting a maximum threshold for the percentage of liquidity that can be removed at any given time or requiring a multi-step process for large withdrawals, possibly including a time delay or the need for multiple approvals. Additionally, introducing mechanisms for gradual liquidity removal could help mitigate sudden market impacts. Implementing these safeguards will help maintain market stability and protect against the potential for manipulation or adverse market reactions due to significant liquidity changes.

LDP - Loss Division Precision

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#LL374,474,574,722,742,769
Status	Unresolved

Description

The contract is designed to calculate the percentage of liquid tokens relative to the circulating supply of the `shadowGoldToken`. However, a precision loss issue occurs due to the use of integer arithmetic for multiplication and division operations. Specifically, when calculating `liquidPercent`, the contract multiplies `liquidTokens` by `10000` before dividing by the `shadowGoldToken.getCirculatingSupply`. If the `liquidTokens` value is significantly smaller than the circulating supply, the multiplication by `10000` may not be sufficient to preserve precision, leading to a situation where the expected nonzero result becomes zero. This issue is exacerbated in cases where the multiplier results in a value that, when divided, is less than 1 due to the lack of floating-point arithmetic in Solidity, resulting in a premature rounding down to 0.

```
uint256 lpOwnershipPercent = (wethPair.balanceOf(address(this)) *
    10000) / wethPair.totalSupply(
uint256 liquidTokens = (shadowGoldToken.balanceOf(
    address(wethPair)
) * lpOwnershipPercent) / 1000
uint256 liquidPercent = ((liquidTokens * 10000) /
    shadowGoldToken.getCirculatingSupply());
```

Recommendation

It is recommended to increase the precision in calculations by using `1e18` as the multiplication factor instead of `10000`. This change significantly enhances the calculation's accuracy by maintaining more significant digits through the multiplication and division processes, thus reducing the risk of premature rounding to zero. By adopting a higher precision constant, the contract can better handle small ratios between `liquidTokens` and the circulating supply, ensuring more accurate and reliable results. Additionally, incorporating well-tested mathematical libraries designed for handling high-precision arithmetic in Solidity can further mitigate potential precision loss and improve the robustness of the contract's calculations.

MEN - Misleading Event Naming

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#L417,464
Status	Unresolved

Description

The contract is using the `burntShadowFi` event to log transactions involving the transfer of `amountWeth` to `wethPair` and the burning of `amountToken` of the `shadowGoldToken`. However, the naming of the event suggests that both the `amountWeth` and `amountToken` are being burned, which is not the case. The `amountWeth` is merely transferred to the `wethPair`, not removed from circulation. This discrepancy between the event's implication and the actual operation performed can lead to confusion and misinterpretation of the contract's actions. The use of burntShadowFi as an event name is thus misleading, as it inaccurately represents the nature of the transactions being logged, particularly the handling of amountWeth and amountPaxg, which are not subjected to a burn mechanism but are instead transferred to their respective pairs.

```
assert(IERC20(weth).transfer(address(wethPair), amountWeth));
wethPair.sync();
shadowGoldToken.burn(amountToken
...
emit burntShadowFi(amountWeth, amountToken);
...
assert(IERC20(paxg).transfer(address(paxgPair), amountPaxg));
paxgPair.sync();
shadowGoldToken.burn(amountToken
...
emit burntShadowFi(amountPaxg, amountToken);
```

Recommendation

It is recommended to revise the event naming and structure to accurately reflect the actions taken by the contract. Specifically, a separate event for token transfers to liquidity pairs and another for token burns should be considered. Otherwise renaming the `burntShadowFi` event to something more descriptive of its actual functionality could clarify the operations being performed. Additionally, introducing parameters within the event or creating separate events to distinctly indicate token transfers and burns would enhance transparency and understanding. This approach would prevent confusion and ensure that the contract's intentions and actions are clearly communicated to developers, auditors, and users alike.

MC - Missing Check

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#L384,482,483,531,583,649,778,799
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues. Specifically, the contract lacks a check to verify that the `liquidPercent` is greater than the `percent` variable. Consequently, if the `liquidPercent` value is lower than `percent`, the calculation will result in an underflow, potentially leading to unexpected behavior.

```
require(
    liquidPercent - percent >= 400,
    "The total percent remaining in liquidity must be at least 4% of the circulating supply."
);
```

Recommendation

The team is advised to properly check the variables according to the required specifications. It is recommended to incorporate additional checks to ensure that the `liquidPercent` is always greater than the `percent` variable to prevent underflow scenarios. This safeguard will maintain the integrity of calculations and prevent potential issues arising from insufficient liquidity percentages.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#L370,417,468,517,569,635
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(
    percent >= 1 && percent <= 9200,
    "Invalid parameter is provided"
);
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#L608,674
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.

```
ISwapRouter.ExactInputParams memory params = ISwapRouter
    .ExactInputParams({
        path: abi.encodePacked(weth, poolFee, usdc, poolFee, paxg),
        recipient: address(this),
        amountIn: amountWeth,
        amountOutMinimum: 0
    }
uint256 amountPaxg = swapRouter.exactInput(params);
...
ISwapRouter.ExactInputParams memory params = ISwapRouter
    .ExactInputParams({
        path: abi.encodePacked(paxg, poolFee, usdc, poolFee, weth),
        recipient: address(this),
        amountIn: amountPaxg,
        amountOutMinimum: 0
    });
uint256 amountWeth = swapRouter.exactInput(params);
```

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.

RED - Redudant Event Declaration

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#L288
Status	Unresolved

Description

The contract uses events that are not emitted within the contract's functions. As a result, these declared events are redundant and serve no purpose within the contract's current implementation.

```
event addedLiquidity(uint256 liquidity);
```

Recommendation

To optimize contract performance and efficiency, it is advisable to regularly review and refactor the codebase, removing the unused event declarations. This proactive approach not only streamlines the contract, reducing deployment and execution costs but also enhances readability and maintainability.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#L278,299
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

The contract is designed to function as a locker for tokens, intending to secure them by locking within its structure. However, the tokens intended to be locked are not transferred to the contract at the time of its initialization. Instead, the contract relies on an external administrator to deposit the tokens post-deployment. This approach introduces a significant risk, as the contract's effectiveness and security are contingent upon the actions of the external administrator. The dependency on an external entity not only centralizes control but also exposes the contract to potential delays, or mismanagement of the tokens, thereby undermining the trust and functionality of the contract.

```
   address private weth =
0x0f4e9Ee7E15A7D135703b7d469E3B18c91D3F1f3;
    address private paxg =
0xA5460F029473D74c8895bA493540E7cd98461316;
    ...
        shadowGoldToken = IShadowGoldToken(
            0x984Da7E52Db4838Aae9a4D7C392619BE4FdD6081
        );
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the presale tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process. It is recommended to modify the contract's initialization process to include the transfer of tokens intended to be locked.

UVF - Unutilized View Functions

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#L704,718,738,760
Status	Unresolved

Description

The contract is utilizing multiple public view functions to retrieve various pieces of information, such as LP ownership percentage, liquid tokens, liquid percent, and the amount to be removed for operations like shadow bursting. However these functions implement similar functionality, essentially calculating and returning values based on the contract's state and given parameters. This redundancy not only increases the contract's complexity but also its deployment and execution cost due to the duplicated logic. Additioanally, it introduces unnecessary points of maintenance and potential inconsistency, as updates to the logic must be meticulously synchronized across all functions.

```
function getLPOwnershipPercent(
        bool wethOrPaxg
    ) public view returns (uint256 lpOwnershipPercent) {
        if (wethOrPaxg) {
            lpOwnershipPercent =
                (wethPair.balanceOf(address(this)) * 10000) /
                wethPair.totalSupply();
        } else {
            lpOwnershipPercent =
                (paxgPair.balanceOf(address(this)) * 10000) /
                paxgPair.totalSupply();
        }
    }

    function getLiquidTokens(
        bool wethOrPaxg
    ) public view returns (uint256 liquidTokens) {
        ...
    }

    function getLiquidPercent(
        bool wethOrPaxg
    ) public view returns (uint256 liquidPercent) {
        ...
        }
    }

    function getShadowBurstAmount(
        uint256 percent,
        bool wethOrPaxg
    ) public view returns (uint256 removeAmount) {
        if (wethOrPaxg) {
        ...
        }
```

Recommendation

It is recommended to consolidate these view functions into a smaller number of versatile functions or internal library calls that can be reused within the contract. This approach would reduce redundancy, simplify the contract's interface, and decrease the potential for inconsistencies in logic updates. Additionally, consider implementing internal helper functions that these public view functions can call, ensuring that the core logic is defined in a single location. This refactoring will not only optimize gas costs for deployments and interactions but also enhance the contract's readability and maintainability.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#L274,278,279,280,282,284
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
uint24 poolFee = 2500
address private weth =
0x0f4e9Ee7E15A7D135703b7d469E3B18c91D3F1f3
address private paxg =
0xA5460F029473D74c8895bA493540E7cd98461316
address private usdc =
0xa36A287Bf83769F9A009E8650D9a9FBfFaF06608

IDEXPair public wethPair =
        IDEXPair(0xbaf048485A899f57039cb5a91b5506E68002b22e)

IDEXPair public paxgPair =
        IDEXPair(0xA8B922432e56D5094507A0a0fe6385B6B5B4DEFF)
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#L287,288,289,290,343,356
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.

```
event burntShadowFi(uint256 amountWethAdded, uint256
amountTokenBurnt);
event addedLiquidity(uint256 liquidity);
event wethInjected(uint256 wethAmount);
event paxgInjected(uint256 paxgAmount);
uint256 _extraLockTime
address _token
```

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.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#L374,376,379,387,421,423,426,434,472,474,477,485,520,522,525,533,572,574,577,585,638,640,643,651,720,722,727,729,740,742,745,748,750,753,767,769,772,780,788,790,793,801
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 lpOwnershipPercent = (paxgPair.balanceOf(address(this))
*
                10000) / paxgPair.totalSupply()
liquidTokens =
                (shadowGoldToken.balanceOf(address(paxgPair)) *
                    lpOwnershipPercent) /
                10000
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#L444,543,662
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 amountToken
uint256 amountPaxg
```

Recommendation

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

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#L56,57,124
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 ^0.7.6;
pragma abicoder v2;
```

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	contracts/ShadowGoldLiquidityLocker.sol#L56,124
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 ^0.7.6;
```

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.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	contracts/ShadowGoldLiquidityLocker.sol#L336,337,365,505,507,553,555
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.

```
wethPair.transfer(owner(), wethPair.balanceOf(address(this)))
paxgPair.transfer(owner(), wethPair.balanceOf(address(this)))
IERC20(_token).transfer(address(msg.sender), amount)
IERC20(weth).transfer(address(wethPair), amountWeth)
shadowGoldToken.transfer(address(paxgPair), amountToken)
IERC20(paxg).transfer(address(paxgPair), amountPaxg)
shadowGoldToken.transfer(address(wethPair), amountToken)
```

Recommendation

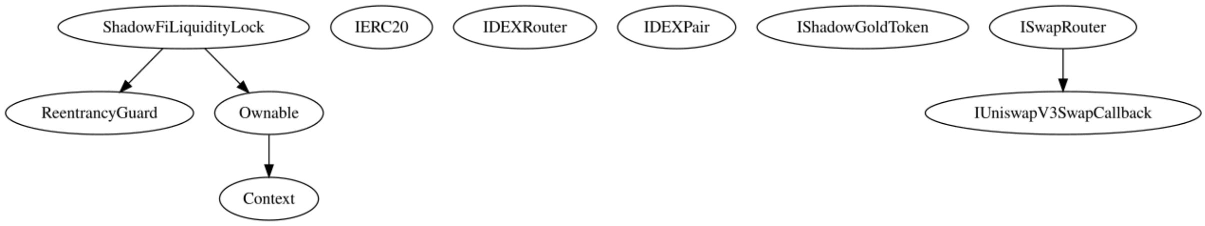
The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Functions Analysis

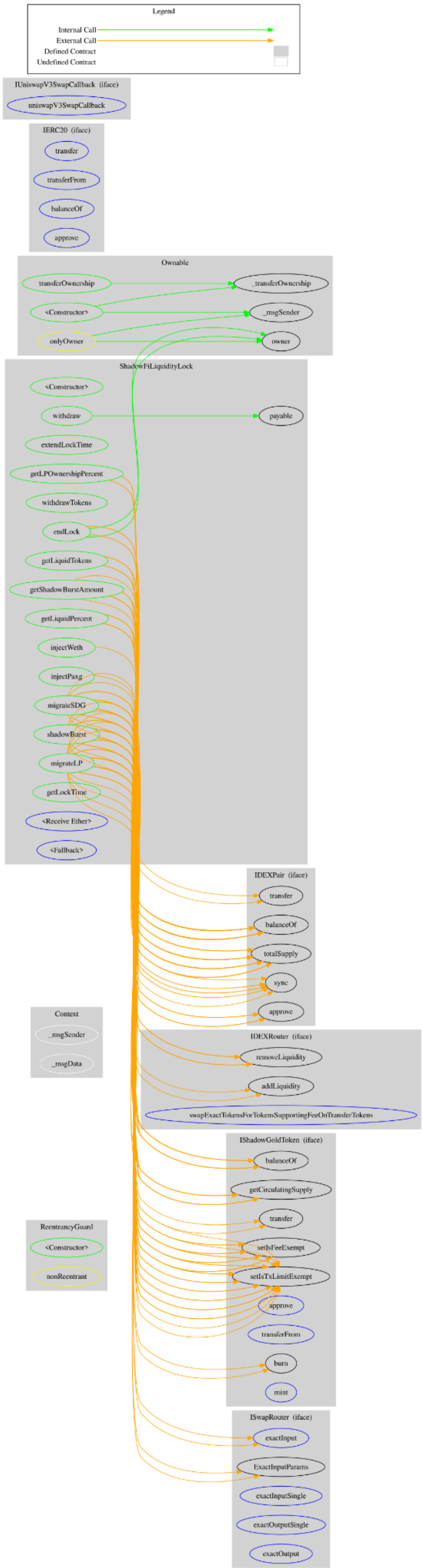
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
ShadowFiLiquidityLock	Implementation	Ownable, ReentrancyGuard		
		Public	✓	-
	endLock	Public	✓	onlyOwner
	extendLockTime	Public	✓	onlyOwner
	withdraw	Public	✓	onlyOwner
	withdrawTokens	Public	✓	onlyOwner
	shadowBurst	Public	✓	onlyOwner
	migrateSDG	Public	✓	onlyOwner
	migrateLP	Public	✓	onlyOwner
	getLPOwnershipPercent	Public		-
	getLiquidTokens	Public		-
	getLiquidPercent	Public		-
	getShadowBurstAmount	Public		-
	injectWeth	Public	✓	onlyOwner
	injectPaxg	Public	✓	onlyOwner
	getLockTime	Public		-
		External	Payable	-
		External	Payable	-

ISwapRouter	Interface	IUniswapV3SwapCallback		
	exactInputSingle	External	Payable	-
	exactInput	External	Payable	-
	exactOutputSingle	External	Payable	-
	exactOutput	External	Payable	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
IUniswapV3SwapCallback (iface)
uniswapV3SwapCallback
IERC20 (iface)
transfer
transferFrom
balanceOf
approve
Ownable
transferOwnership
_transferOwnership
<Constructor>
_msgSender
onlyOwner
owner
ShadowFiLiquidityLock
<Constructor>
withdraw
payable
extendLockTime
getLPOwnershipPercent
withdrawTokens
endLock
getLiquidTokens
getShadowBurstAmount
getLiquidPercent
injectWeth
injectPaxg
migrateSDG
shadowBurst
migrateLP
getLockTime
<Receive Ether>
<Fallback>
IDEXPair (iface)
transfer
balanceOf
totalSupply
sync
approve
Context
_msgSender
_msgData
IDEXRouter (iface)
removeLiquidity
addLiquidity
swapExactTokensForTokensSupportingFeeOnTransferTokens
IShadowGoldToken (iface)
balanceOf
getCirculatingSupply
transfer
setIsFeeExempt
setIsTxLimitExempt
approve
transferFrom
burn
mint
ReentrancyGuard
<Constructor>
nonReentrant
ISwapRouter (iface)
exactInput
ExactInputParams
exactInputSingle
exactOutputSingle
exactOutput

Summary

ShadowFi contract implements a locker mechanism. The `ShadowFiLiquidityLock` contract manages the liquidity of the `ShadowGoldToken`, providing mechanisms for locking liquidity, managing funds, migrating liquidity, and adjusting the token's supply. Its functionalities are designed to enhance the stability and trustworthiness of the token within the DeFi ecosystem. This audit investigates security issues, business logic concerns and potential improvements.

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