Cyberscope

Audit Report

PigTWAP

July 2023

SHA256 cbdee9ed3f049fce188a4c5177e31f96f6936e684c44d8a34f105f7f29f69715

Audited by © cyberscope

Table of Contents

Review

Testing Deploy	https://testnet.bscscan.com/address/0x1f7bd6fd16d905883f4638654803bae43bb27eb3

Audit Updates

Initial Audit	08 Jul 2023

Source Files

Filename	SHA256
contracts/PigTWAP.sol	cbdee9ed3f049fce188a4c5177e31f96f6936e684c44d8a34f105f7f29f69715

Overview

The PigTWAP contract provides a way to calculate the average price of a token pair on Uniswap over time, which can be useful for various applications such as automated trading, price monitoring, and liquidity provision.

Findings Breakdown

Critical	0
Medium	0
Minor / Informative	11

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

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	DPI	Decimals Precision Inconsistency	Unresolved
	CR	Code Repetition	Unresolved
	RM	Redundant Multiplication	Unresolved
	IDI	Immutable Declaration Improvement	Unresolved
	L02	State Variables could be Declared Constant	Unresolved
	L04	Conformance to Solidity Naming Conventions	Unresolved
	L09	Dead Code Elimination	Unresolved
	L13	Divide before Multiply Operation	Unresolved
	L14	Uninitialized Variables in Local Scope	Unresolved
	L16	Validate Variable Setters	Unresolved
	L18	Multiple Pragma Directives	Unresolved

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	contracts/PigTWAP.sol#L558,611,613
Status	Unresolved

Description

The decimals field of a contract's ERC20 token can be used to specify the number of decimal places that the token uses. For example, if decimals are set to `8`, it means that the smallest unit of the token is `0.00000001`, and if decimals are set to `18`, it means that the smallest unit of the token is `0.000000000000000001`.

However, there is an inconsistency in the way that the decimals field is handled in some ERC20 contracts. The ERC20 specification does not specify how the decimals field should be implemented, and as a result, some contracts use different precision numbers.

This inconsistency can cause problems when interacting with these contracts, as it is not always clear how the decimals field should be interpreted. For example, if a contract expects the decimals field to be 18 digits, but the contract being interacted with uses 8 digits, the result of the interaction may not be what was expected.

```
uint multiplier = 1*10**18; // TWAP per single token
..
amountOut = price0Average.mul(multiplier).decode144();
```

Recommendation

To avoid these issues, it is important to carefully review the implementation of the decimals field of the underlying tokens. The team is advised to normalize each decimal to one single source of truth. A recommended way is to scale all the decimals to the greatest token's decimal. Hence, the contract will not lose precision in the calculations.

The following example depicts 3 tokens with different decimals precision.

ERC20	Decimals
Token 1	6
Token 2	9
Token 3	18

All the decimals could be normalized to 18 since it represents the ERC20 token with the greatest digits.

CR - Code Repetition

Criticality	Minor / Informative
Location	contracts/PigTWAP.sol#L592,595
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.

```
price0Average = FixedPoint.uq112x112(
    uint224((price0Cumulative - price0CumulativeLast) / timeElapsed)
);
price1Average = FixedPoint.uq112x112(
    uint224((price1Cumulative - price1CumulativeLast) / timeElapsed)
);
```

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.

RM - Redundant Multiplication

Criticality	Minor / Informative
Location	contracts/PigTWAP.sol#L558
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

Multiplication by 1 is unnecessary because it does not change the value of the number being multiplied. When you multiply a number by 1, the result is always the same as the original number. Therefore, it has no effect on the final outcome or computation.

```
uint multiplier = 1*10**18;
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	contracts/PigTWAP.sol#L567
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.

```
auth
```

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.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	contracts/PigTWAP.sol#L558
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.

```
uint multiplier = 1*10**18
```

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/PigTWAP.sol#L232,238,387,388,405
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.

```
struct uq112x112 {
        uint224 _x;
    }

struct uq144x112 {
        uint256 _x;
    }
function DOMAIN_SEPARATOR() external view returns (bytes32);
function PERMIT_TYPEHASH() external pure returns (bytes32);
function MINIMUM_LIQUIDITY() external pure returns (uint);
```

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.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	contracts/PigTWAP.sol#L30,67,121,248,253,258,277,285,313,349,357,433,440,451,458,465,475,484,495
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 mostSignificantBit(uint256 x) internal pure returns (uint8 r) {
        require(x > 0, 'BitMath::mostSignificantBit: zero');

        if (x >= 0x100000000000000000000000000000000) {
            x >>= 128;
            r += 128;
...
        }
        if (x >= 0x4) {
            x >>= 2;
            r += 2;
        }
        if (x >= 0x2) r += 1;
    }

...
```

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.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	contracts/PigTWAP.sol#L187,188,191,192,193,194,195,196,197,198,199
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.

```
d /= pow2
r *= 2 - d * r
```

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/PigTWAP.sol#L488
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.

```
uint i
```

Recommendation

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

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	contracts/PigTWAP.sol#L567
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
auth = _auth
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	contracts/PigTWAP.sol#L4,25,114,169,223,370,425,508,544
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 =0.6.6;
pragma solidity >=0.5.0;
pragma solidity >=0.4.0;
pragma solidity 0.6.6;
```

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.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
SafeMath	Library			
	add	Internal		
	sub	Internal		
	mul	Internal		
BitMath	Library			
	mostSignificantBit	Internal		
	leastSignificantBit	Internal		
Babylonian	Library			
	sqrt	Internal		
FullMath	Library			
	fullMul	Internal		
	fullDiv	Private		
	mulDiv	Internal		
FixedPoint	Library			

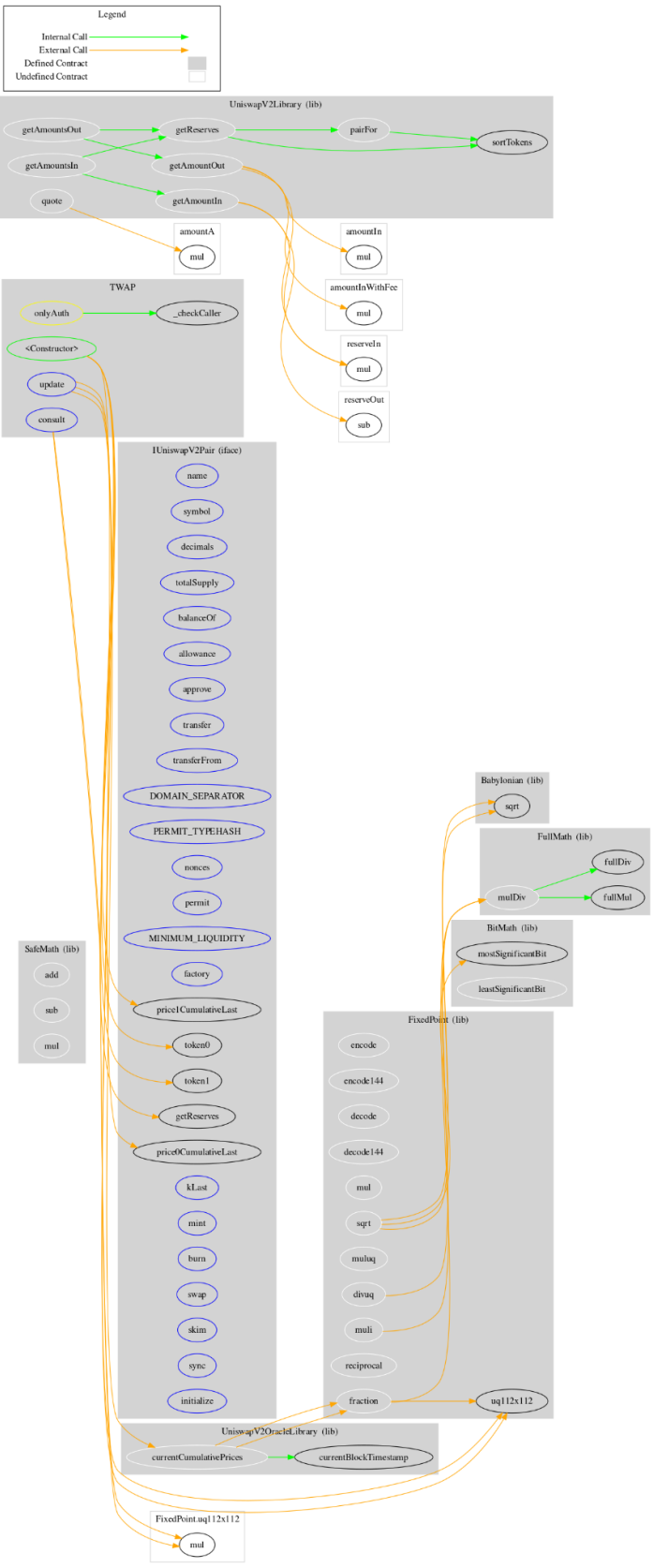
	encode	Internal		
	encode144	Internal		
	decode	Internal		
	decode144	Internal		
	mul	Internal		
	muli	Internal		
	muluq	Internal		
	divuq	Internal		
	fraction	Internal		
	reciprocal	Internal		
	sqrt	Internal		
IUniswapV2Pair	Interface			
	name	External		-
	symbol	External		-
	decimals	External		-
	totalSupply	External		-
	balanceOf	External		-
	allowance	External		-
	approve	External	✓	-
	transfer	External	✓	-
	transferFrom	External	✓	-
	DOMAIN_SEPARATOR	External		-

	PERMIT_TYPEHASH	External		-
	nonces	External		-
	permit	External	✓	-
	MINIMUM_LIQUIDITY	External		-
	factory	External		-
	token0	External		-
	token1	External		-
	getReserves	External		-
	price0CumulativeLast	External		-
	price1CumulativeLast	External		-
	kLast	External		-
	mint	External	✓	-
	burn	External	✓	-
	swap	External	✓	-
	skim	External	✓	-
	sync	External	✓	-
	initialize	External	✓	-
UniswapV2Library	Library			
	sortTokens	Internal		
	pairFor	Internal		
	getReserves	Internal		
	quote	Internal		

	getAmountOut	Internal		
	getAmountIn	Internal		
	getAmountsOut	Internal		
	getAmountsIn	Internal		
UniswapV2OracleLibrary	Library			
	currentBlockTimestamp	Internal		
	currentCumulativePrices	Internal		
TWAP	Implementation			
		Public	✓	-
	_checkCaller	Internal		
	update	External	✓	onlyAuth
	consult	External		-

Inheritance Graph

Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
UniswapV2Library (lib)
getAmountsOut
getReserves
pairFor
sortTokens
getAmountsIn
getAmountOut
quote
getAmountIn
amountA
mul
amountIn
mul
amountInWithFee
mul
reserveIn
mul
reserveOut
sub
TWAP
onlyAuth
_checkCaller
<Constructor>
update
consult
IUniswapV2Pair (iface)
name
symbol
decimals
totalSupply
balanceOf
allowance
approve
transfer
transferFrom
DOMAIN_SEPARATOR
PERMIT_TYPEHASH
nonces
permit
MINIMUM_LIQUIDITY
factory
price1CumulativeLast
token0
token1
getReserves
price0CumulativeLast
kLast
mint
burn
swap
skim
sync
initialize
SafeMath (lib)
add
sub
mul
Babylonian (lib)
sqrt
FullMath (lib)
fullDiv
mulDiv
fullMul
BitMath (lib)
mostSignificantBit
leastSignificantBit
FixedPoint (lib)
encode
encode144
decode
decode144
mul
sqrt
muluq
divuq
muli
reciprocal
fraction
uq112x112
UniswapV2OracleLibrary (lib)
currentCumulativePrices
currentBlockTimestamp
FixedPoint.uq112x112
mul

Summary

ProfitPIG contract implements a utility mechanism. 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