Cyberscope

Audit Report

DXS Bridge

March 2023

GitHub	https://github.com/OlegPanagushin/dxs-eth-bridge
Commit	0abd2b7bad52300c29302ec7e6945f20f7fc12bb
Audited by	© cyberscope

Table of Contents

Review

Repository	https://github.com/OlegPanagushin/dxs-eth-bridge
Commit	0abd2b7bad52300c29302ec7e6945f20f7fc12bb

Audit Updates

Initial Audit	03 Apr 2023

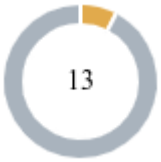
Source Files

Filename	SHA256
LiquidityPool.sol	4bd46a05651051f308e71532b265836885965d67aebb0b5a3db9ee96975062f9

Introduction

The LiquidityPool contract allows users to deposit and withdraw ERC20 tokens. The contract consists of three components: LiquidityProvider, Treasury, and LiquidityPool. The LiquidityProvider generates deposit addresses for users, Treasury transfers tokens out of the LiquidityPool, and LiquidityPool holds the deposited tokens and allows users to deposit and withdraw them.

Findings Breakdown



Critical	0
Medium	1
Minor / Informative	12

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	1	0	0	0
Minor / Informative	12	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	CR	Code Reusability	Unresolved
	IAA	Insuficient Approval Amount	Unresolved
	DPI	Decimals Precision Inconsistency	Unresolved
	RFC	Redundant Function Call	Unresolved
	PSU	Potential Subtraction Underflow	Unresolved
	MEM	Misleading Error Messages	Unresolved
	RC	Redundant Check	Unresolved
	RSK	Redundant Storage Keyword	Unresolved
	IDI	Immutable Declaration Improvement	Unresolved
	L02	State Variables could be Declared Constant	Unresolved
	L05	Unused State Variable	Unresolved
	L16	Validate Variable Setters	Unresolved
	L19	Stable Compiler Version	Unresolved

CR - Code Reusability

Criticality	Minor / Informative
Location	LiquidityPool.sol#L305
Status	Unresolved

Description

The code segment `keccak256(abi.encodePacked(withdrawAddress, tokenType))` is the same as the one returned from the `getSalt` function.

```
account = _accounts[
    keccak256(abi.encodePacked(withdrawAddress, tokenType))
];
```

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.

IAA - Insuficient Approval Amount

Criticality	Medium
Location	LiquidityPool.sol#L22
Status	Unresolved

Description

The contract uses an arbitary large value as the max approval amount. A token's max supply can possibly be greater than this value. As a result, the approval amount will be insufficient.

```
require(
    IErc20Min(token).approve(
        msg.sender,
        999_999_999_999_999_999 * 10**decimals
    ), // quiet large amount
    "Not approved"
);
```

Recommendation

The team is advised to use the maximum value of a 256-bit integer as the approval amount to avoid this issue.

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	LiquidityPool.sol#L18
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 is 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.

```
contract LiquidityProvider {
    constructor(address token, uint8 decimals) {
        require(
            IErc20Min(token).approve(
                msg.sender,
                999_999_999_999_999_999 * 10**decimals
            ), // quiet large amount
            "Not approved"
        );
    }
}
```

Recommendation

To avoid these issues, it is recommended to get the decimals from the token's contract instead of passing it as an argument. This way, the contract will ensure that the decimals are consistent for each token.

RFC - Redundant Function Call

Criticality	Minor / Informative
Location	LiquidityPool.sol#L227
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.

The contract calls the `max` function to determine which of the `balance` and `withdrawLimitCents` variables is greater. This comparison has already been made at the first if-block. As a result, calling the `max` function is redundant.

```
if (balance > account.withdrawLimitCents) {
    if (account.withdrawLimitCents < _maxWithdrawAmountCents) {
        account.withdrawLimitCents = min(
            max(account.withdrawLimitCents, balance),
            _maxWithdrawAmountCents
        );
    }
}
```

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.

PSU - Potential Subtraction Underflow

Criticality	Minor / Informative
Location	LiquidityPool.sol#L328
Status	Unresolved

Description

The contract subtracts two values, the second value may be greater than the first value if the contract owner misuses the configuration. As a result, the subtraction may underflow and cause the execution to revert.

The `centsToToken` function converts an amount to the appropriate token amount, based on its decimals. The token decimals can be set to a value that is greater or equal to 1. If the value is 1, then the operation will lead to an underflow.

```
function centsToToken(uint256 cents, uint8 tokenType)
    private
    view
    returns (uint256)
{
    return cents * (10**(_tokens[tokenType].decimals - 2));
}
```

Recommendation

The team is advised to properly handle the code to avoid underflow subtractions and ensure the reliability and safety of the contract. The contract should ensure that the first value is always greater than the second value. It should add a sanity check in the setters of the variable or not allow executing the corresponding section if the condition is violated.

MEM - Misleading Error Messages

Criticality	Minor / Informative
Location	LiquidityPool.sol#L212
Status	Unresolved

Description

The contract is using misleading error messages. These error messages do not accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(_tokens[tokenType].decimals > 0, "Unknown");
```

Recommendation

The team is advised to carefully review the source code in order to address these issues. To accelerate the debugging process and mitigate these issues, the team should use more specific and descriptive error messages.

RC - Redundant Check

Criticality	Minor / Informative
Location	LiquidityPool.sol#L212,316
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.

The `_tokens[tokenType].decimals > 0` expression is checked twice in the `collect` function. As a result, the second require check is redundant.

The contract calls the `createAccount` function only if the `account.depositAddress == address(0)` expression is true. As a result, the require check is redundant.

```
require(_tokens[tokenType].decimals > 0, "Unknown");
...
if (newAccount) {
    createAccount(withdrawAddress, tokenType);
}
...
require(account.depositAddress == address(0), "Existig account");
```

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.

RSK - Redundant Storage Keyword

Criticality	Minor / Informative
Location	LiquidityPool.sol#L303
Status	Unresolved

Description

The contract uses the `storage` keyword in a view function. The `storage` keyword is used to persist data on the contract's storage. View functions are functions that do not modify the state of the contract and do not perform any actions that cost gas (such as sending a transaction). As a result, the use of the `storage` keyword in view functions is redundant.

```
Account storage account
```

Recommendation

It is generally considered good practice to avoid using the `storage` keyword in view functions, because it is unnecessary and can make the code less readable.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	LiquidityPool.sol#L110,111,113
Status	Unresolved

Description

The contract is using variables that initialize them only in the constructor. The other functions are not mutating the variables. These variables are not defined as `immutable`.

```
_withdrawResetTimeout
_maxWithdrawAmountCents
_treasury
```

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	LiquidityPool.sol#L88,89,90
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.

```
address private _usdtToken
address private _usdcToken
address private _daiToken
```

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.

L05 - Unused State Variable

Criticality	Minor / Informative
Location	LiquidityPool.sol#L88,89,90
Status	Unresolved

Description

An unused state variable is a state variable that is declared in the contract, but is never used in any of the contract's functions. This can happen if the state variable was originally intended to be used, but was later removed or never used.

Unused state variables can create clutter in the contract and make it more difficult to understand and maintain. They can also increase the size of the contract and the cost of deploying and interacting with it.

```
address private _usdtToken
address private _usdcToken
address private _daiToken
```

Recommendation

To avoid creating unused state variables, it's important to carefully consider the state variables that are needed for the contract's functionality, and to remove any that are no longer needed. This can help improve the clarity and efficiency of the contract.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	LiquidityPool.sol#L109,144
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.

```
_nextOwnerAddress = nextOwner
```

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.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	LiquidityPool.sol#L3
Status	Unresolved

Description

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

```
pragma solidity ^0.8.0;
```

Recommendation

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

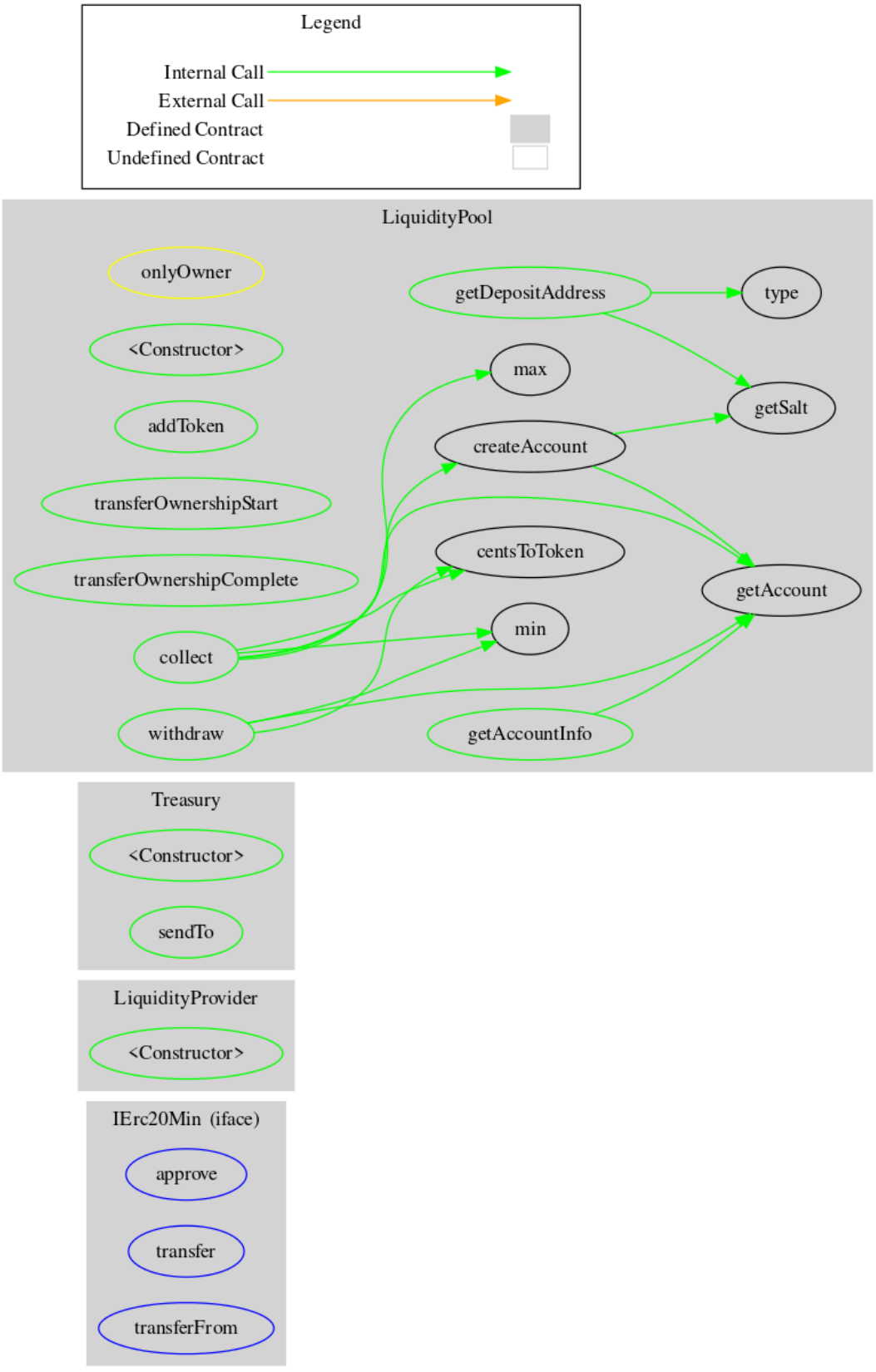
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IErc20Min	Interface			
	approve	External	✓	-
	transfer	External	✓	-
	transferFrom	External	✓	-
LiquidityProvider	Implementation			
		Public	✓	-
Treasury	Implementation			
		Public	✓	-
	sendTo	Public	✓	-
LiquidityPool	Implementation			
		Public	✓	-
	addToken	Public	✓	onlyOwner
	transferOwnershipStart	Public	✓	onlyOwner
	transferOwnershipComplete	Public	✓	-
	getAccountInfo	Public		-

	getDepositAddress	Public		-
	collect	Public	✓	onlyOwner
	withdraw	Public	✓	onlyOwner
	getAccount	Private		
	createAccount	Private	✓	
	centsToToken	Private		
	getSalt	Private		
	max	Private		
	min	Private		

Inheritance Graph

Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
LiquidityPool
onlyOwner
<Constructor>
addToken
transferOwnershipStart
transferOwnershipComplete
collect
withdraw
getDepositAddress
type
max
getSalt
createAccount
centsToToken
getAccount
min
getAccountInfo
Treasury
<Constructor>
sendTo
LiquidityProvider
<Constructor>
IErc20Min (iface)
approve
transfer
transferFrom

Summary

DXS Bridge 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