Cyberscope

Audit Report

Dextop Staking

March 2024

Network	ETH
Address	0xb058a163592bdee2159B7655D7150B9A2E749EC2
Audited by	© cyberscope

Table of Contents

Review

Explorer	https://etherscan.io/address/0xb058a163592bdee2159b7655d7150b9a2e749ec2

Audit Updates

Initial Audit	15 Mar 2024

Source Files

Filename	SHA256
contracts/DXTSteak.sol	ee8c7a173c2e8618d22a3ed2c392037a1a88a61e0cf6d0c6edbbf59613a22ba3
@openzeppelin/contracts/utils/Context.sol	847fda5460fee70f56f4200f59b82ae622bb03c79c77e67af010e31b7e2cc5b6
@openzeppelin/contracts/utils/Address.sol	b3710b1712637eb8c0df81912da3450da6ff67b0b3ed18146b033ed15b1aa3b9
@openzeppelin/contracts/token/ERC20/IERC20.sol	6f2faae462e286e24e091d7718575179644dc60e79936ef0c92e2d1ab3ca3cee
@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol	471157c89111d7b9eab456b53ebe9042bc69504a64cb5cc980d38da9103379ae
@openzeppelin/contracts/token/ERC20/extensions/IERC20Permit.sol	912509e0e9bf74e0f8a8c92d031b5b26d2d35c6d4abf3f56251be1ea9ca946bf
@openzeppelin/contracts/security/ReentrancyGuard.sol	fa97ea556c990ee44f2ef4c80d4ef7d0af3f5f9b33a0214291114068810 6f5a9
@openzeppelin/contracts/access/Ownable.sol	38578bd71c0a909840e67202db527cc6b4e6b437e0f39f0c909da32c1e30cb81

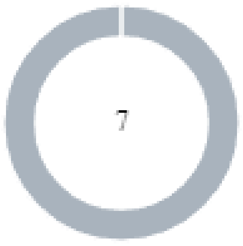
Overview

The "DXTSteak" stakingsmart contract is designed for staking "DXT" tokens. It offers functionalities for users to stake, unlock, and add more tokens to their existing stake. Staking initiates with the `lockTokens` function, requiring a minimum amount and ensuring no active stake exists. The `unlockTokens` function allows users to retrieve their staked tokens after a set period, while `addToStake` enables adding tokens to an active stake.

The contract tracks each staker's details, including active status, locked amount, and claim history, through the `StakerInfo` struct. It also calculates and distributes staking rewards with a `claimDxt` function, based on each user's share of the total staked pool and adhering to a claim interval.

Additionally, it supports batch retrieval of staker addresses for efficient data handling and integrates OpenZeppelin's standards for security and ownership management.

Findings Breakdown



●	Critical	0
●	Medium	0
●	Minor / Informative	7

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	0	0	0	0
●	Minor / Informative	7	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	BLC	Business Logic Concern	Unresolved
●	CO	Code Optimization	Unresolved
●	CR	Code Repetition	Unresolved
●	MU	Modifiers Usage	Unresolved
●	PTAI	Potential Transfer Amount Inconsistency	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved

BLC - Business Logic Concern

Criticality	Minor / Informative
Location	contracts/DXTSteak.sol#L78,130
Status	Unresolved

Description

The contract allows users to stake tokens by calling `lockTokens` function and add additional tokens to their stake with `addToStake` fucntion without resetting the `lastClaimTime` during the latter. This mechanism permits users to significantly increase their stake just before the end of the claim interval, allowing them to claim rewards on the new, larger amount without the added tokens needing to be staked for the full claim period. While this does not directly compromise the security or functionality of the contract, it introduces a potential for gaming the system that may not align with the intended fair distribution of rewards.

```
function lockTokens(uint256 amount) external nonReentrant {
    ...

    staker.lockedAmount += amount;
    staker.lastStakeTime = block.timestamp;
    staker.lastClaimTime = block.timestamp;
    totalLockedBalance += amount;

    ...
}

function claimDxt() external nonReentrant {
    StakerInfo storage staker = stakersInfo[msg.sender];
    require(staker.isActive, "No active stake found");
    require(
         block.timestamp >= staker.lastClaimTime +
CLAIM_INTERVAL,
        "Claiming too soon"
    );

    ...
```

Recommendation

It is recommended to consider the implications of the current reward claim logic of the reward distribution system. If the intent is to ensure that rewards are proportionate to both the amount staked and the duration it has been staked, then adjusting the logic to reset the `lastClaimTime` upon any addition to an existing stake could mitigate potential gaming of the reward system.

CO - Code Optimization

Criticality	Minor / Informative
Location	contracts/DXTSteak.sol#L81
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. Specifically, the `lockTokens` function uses a `require` statement to ensure that a user cannot stake if they already have a locked amount, explicitly prohibiting multiple active stakes per user. Following this, the function checks the `isActive` flag and the `lockedAmount` again before modifying the user's staking status and updating the total active stakers.

First of all, the `require` statement at the beginning of the `lockTokens` function already ensures that `staker.lockedAmount == 0`, making any subsequent check for `staker.lockedAmount == 0` within the same function redundant. The initial `require` effectively guarantees that any further operations occur under the condition that the user does not have an existing stake.

Secondly, the use of the `isActive` flag to track whether a user has an active stake is redundant. The contract's logic inherently ties a user's staking status to their `lockedAmount`, since a user is considered to be actively staking if and only if `staker.lockedAmount > 0`. Thus, the `isActive` flag's purpose overlaps entirely with the condition checked by `lockedAmount`, rendering it unnecessary.

```
require(
    staker.lockedAmount == 0,
    "User can only have one active stake"
);

if (!staker.isActive) {
    if (staker.lockedAmount == 0) {
        allStakers.push(msg.sender);
    }
    staker.isActive = true;
    totalActiveStakers += 1;
}
```

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.

CR - Code Repetition

Criticality	Minor / Informative
Location	contracts/DXTSteak.sol#L78,130
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 `lockTokens` and `addToStake` functions have overlapping functionality. Both functions are designed to increase the `lockedAmount` for a staker, update the last stake time, adjust the total locked balance in the contract, and perform a safe transfer of tokens from the user to the contract. The primary difference lies in the `lockTokens` function's additional logic to add new stakers to the `allStakers` array and increment the `totalActiveStakers` counter, actions that are not repeated in the `addToStake` function. This redundancy suggests that the contract could be optimized by consolidating these two functions into a single function capable of handling both initial staking and the addition of funds to an existing stake, thereby streamlining the contract's interface and reducing the codebase's complexity.

```
require(additionalAmount >= 1e18, "Minimum addition is 1 DXT");
StakerInfo storage staker = stakersInfo[msg.sender];
require(staker.isActive, "No active stake found");

staker.lockedAmount += additionalAmount;
staker.lastStakeTime = block.timestamp;
totalLockedBalance += additionalAmount;

DXT.safeTransferFrom(msg.sender, address(this),
additionalAmount);
emit DxtLocked(msg.sender, additionalAmount);
```

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.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	contracts/DXTSteak.sol#L79,106,131,153
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(amount >= 1e18, "Minimum stake is 1 DXT");

require(amount >= 1e18, "Minimum unlock is 1 DXT");

...
```

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.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	contracts/DXTSteak.sol#L99,124
Status	Unresolved

Description

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

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

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

```
DXT.safeTransferFrom(msg.sender, address(this), amount);
DXT.safeTransfer(msg.sender, amount);
```

Recommendation

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

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

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

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/DXTSteak.sol#L56,71
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.

```
IERC20 public DXT
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.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	contracts/DXTSteak.sol#L213
Status	Unresolved

Description

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

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

```
assembly {
                mstore(stakersBatch, count) // Resize the
dynamic array to fit the count
            }
```

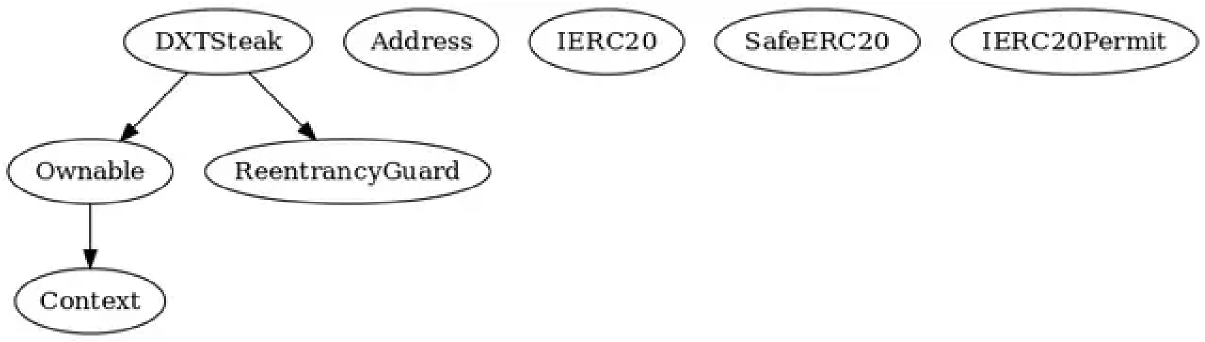
Recommendation

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

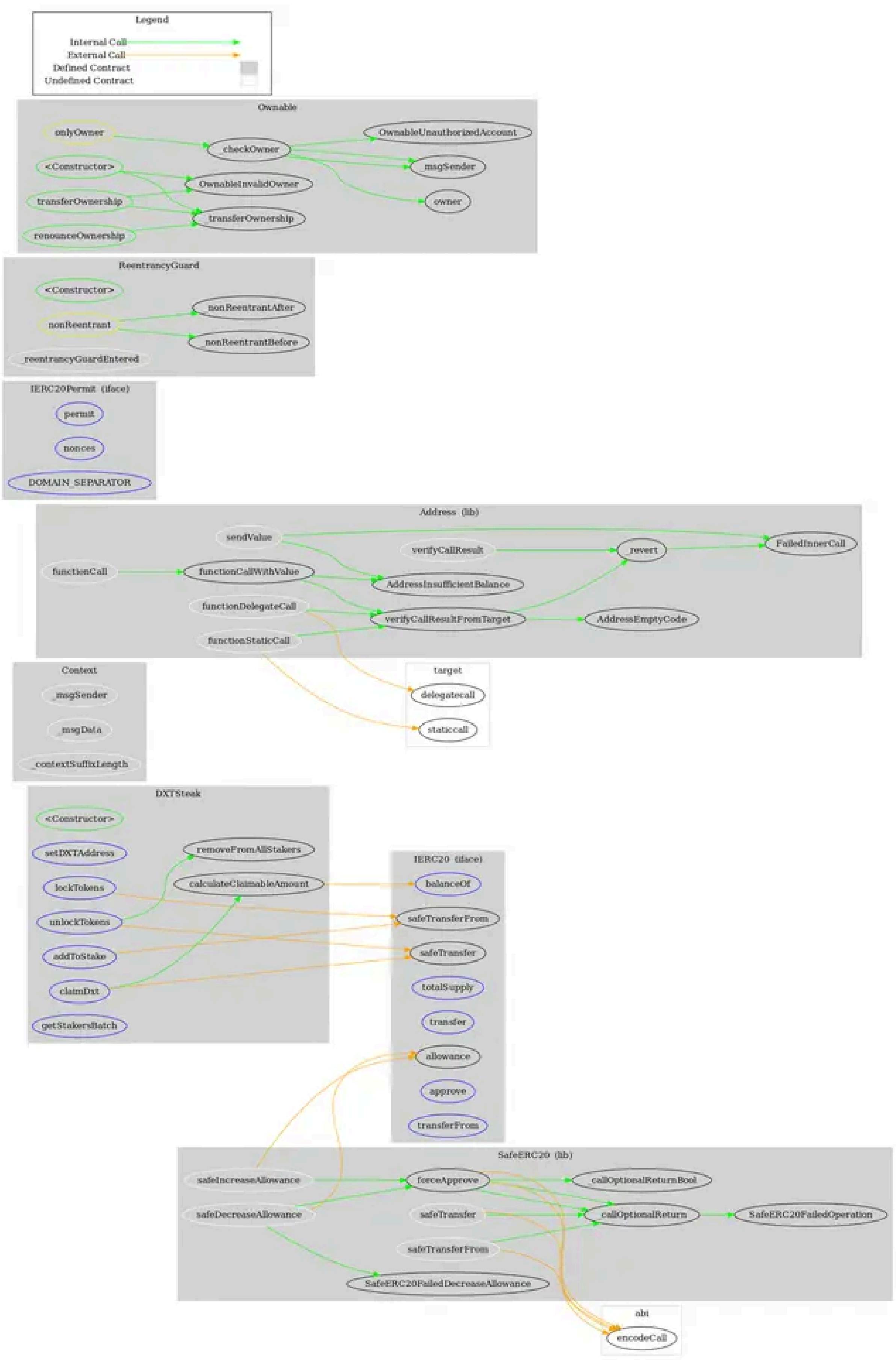
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
DXTSteak	Implementation	Ownable, ReentrancyGuard		
		Public	✓	Ownable
	setDXTAddress	External	✓	onlyOwner
	lockTokens	External	✓	nonReentrant
	unlockTokens	External	✓	nonReentrant
	addToStake	External	✓	nonReentrant
	claimDxt	External	✓	nonReentrant
	calculateClaimableAmount	Public		-
	getStakersBatch	External		-
	removeFromAllStakers	Private	✓	

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Ownable
onlyOwner
_checkOwner
OwnableUnauthorizedAccount
<Constructor>
_msgSender
OwnableInvalidOwner
owner
transferOwnership
_transferOwnership
renounceOwnership
ReentrancyGuard
<Constructor>
_nonReentrantAfter
nonReentrant
_nonReentrantBefore
_reentrancyGuardEntered
IERC20Permit (iface)
permit
nonces
DOMAIN_SEPARATOR
Address (lib)
sendValue
FailedInnerCall
verifyCallResult
_revert
functionCall
functionCallWithValue
AddressInsufficientBalance
functionDelegateCall
verifyCallResultFromTarget
AddressEmptyCode
functionStaticCall
Context
_msgSender
_msgData
_contextSuffixLength
target
delegatecall
staticcall
DXTSteak
<Constructor>
setDXTAddress
removeFromAllStakers
lockTokens
calculateClaimableAmount
unlockTokens
addToStake
claimDxt
getStakersBatch
IERC20 (iface)
balanceOf
safeTransferFrom
safeTransfer
totalSupply
transfer
allowance
approve
transferFrom
SafeERC20 (lib)
safeIncreaseAllowance
forceApprove
_callOptionalReturnBool
safeDecreaseAllowance
safeTransfer
_callOptionalReturn
SafeERC20FailedOperation
safeTransferFrom
SafeERC20FailedDecreaseAllowance
abi
encodeCall

Summary

Dextop staking contract implements a staking and rewards mechanism. This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

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