SCRL

Full Audit Report

DoubleUp Staking Security Assessment

DoubleUp Staking Security Assessment

FULL AUDIT REPORT

Security Assessment by SCRL on **Wednesday, June 19, 2024**

SCRL is deliver a security solution for Web3 projects by expert security researchers.

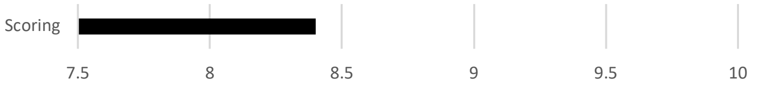
Executive Summary

For this security assessment, SCRL received a request on Sunday, May 5, 2024

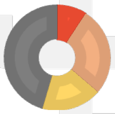
Client	Language	Audit Method	Confidential	Network Chain	Contract
DoubleUp Staking	**Solidity**	**Whitebox**	**Public**	**Polygon**	**0xC1171f874d6777aD54d6bab50dDCC425B58Db4C3**

Report Version	Twitter	Telegram	Website
1.3	**https://twitter.com/doubleup_org**	**https://t.me/doubleup_org**	**https://doubleup.org/**

Scoring:



Vulnerability Summary



18	6	11	1	6	0
Total Findings	Unresolved	Resolved	Mitigate	Acknowledge	Decline

▪	0	Critical		Critical severity is assigned to security vulnerabilities that pose a severe threat to the smart contract and the entire blockchain ecosystem.
▪	1	High	1 Resolved	High-severity issues should be addressed quickly to reduce the risk of exploitation and protect users' funds and data.
▪	3	Medium	1 Mitigate, 2 Resolved	It's essential to fix medium-severity issues in a reasonable timeframe to enhance the overall security of the smart contract.
▪	2	Low	2 Resolved	While low-severity issues can be less urgent, it's still advisable to address them to improve the overall security posture of the smart contract.
▪	0	Very Low		Very Low severity is used for minor security concerns that have minimal impact and are generally of low risk.
▪	4	Informational	2 Unresolved, 2 Resolved	Used to categorize security findings that do not pose a direct security threat to the smart contract or its users. Instead, these findings provide additional information, recommendations
▪	8	Gas-optimization	4 Unresolved, 4 Resolved	Suggestions for more efficient algorithms or improvements in gas usage, even if the current code is already secure.

Audit Scope:

File	SHA-1 Hash
src/DBLUStaking.sol	46d6867a0808e5072065d76732d1e202adbf039c

Audit Version History:

Version	Date	Description
1.0	Tuesday, May 7, 2024	Preliminary Report
1.1	Saturday, 25 May R 2024	Update New Code for re-assessment
1.2	Sunday, June 2, 2024	Update with re-assessment on github commit 65b12b8f9d33a78bd88f9f1306beb64a30698d90
1.3	Wednesday, June 19, 2024	Update with re-assessment on deloyed contract address 0xC1171f874d6777aD54d6bab50dDCC425B58Db4C3

Audit information:

Request Date	Audit Date	Re-assessment Date
Sunday, May 5, 2024	Tuesday, May 7, 2024	Wednesday, 19 June R 2024

Smart Contract Audit Summary

SCRL

SCRL has assessed
the security of this smart contract.

The results of the security
assessment revealed

No Critical Vulnerabilities.

Full Audit Report by SCRL on June 19, 2024

Security Assessment Author

Auditor:	**Mark K.**	[Security Researcher | Redteam]
	Kevin N.	[Security Researcher | Web3 Dev]
	Yusheng T.	[Security Researcher | Incident Response]
Document Approval:	**Ronny C.**	CTO & Head of Security Researcher
	Chinnakit J.	CEO & Founder

Digital Sign

SCRL

Digitally signed by Chinnakit J.
DN: cn=Chinnakit J., o=SCRL,
ou=CEO & Founder,
email=contact@scrl.io, c=TH
Date: 2024.06.19 19:49:44 +07'00'

Disclaimer

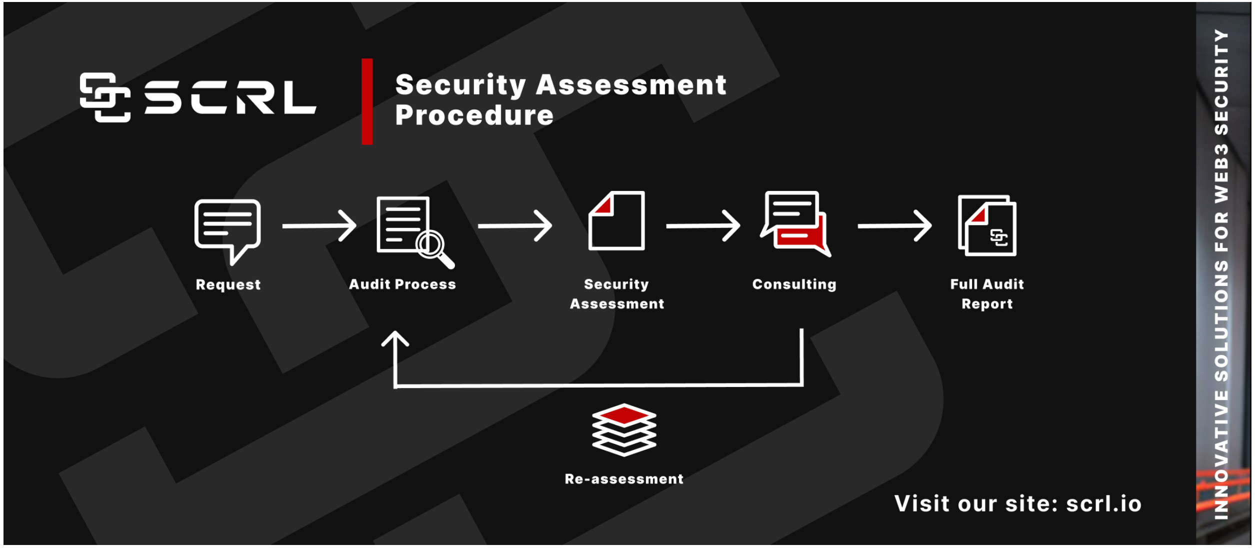
Regarding this security assessment, there are no guarantees about the security of the program instruction received from the client is hereinafter referred to as “**Source code**”.
And **SCRL** hereinafter referred to as “**Service Provider**”, the **Service Provider** will not be held liable for any legal liability arising from errors in the security assessment. The responsibility will be the responsibility of the **Client**, hereinafter referred to as “**Service User**” and the
Service User agrees not to be held liable to the **service provider** in any case. By contract
Service Provider to conduct security assessments with integrity with professional ethics, and transparency to deliver security assessments to users The **Service Provider** has the right to postpone the delivery of the security assessment. If the security assessment is delayed whether caused by any reason and is not responsible for any delayed security assessments.
If **the service provider** finds a vulnerability The **service provider** will notify the **service user** via the Preliminary Report, which will be kept confidential for security. The **service provider** disclaims responsibility in the event of any attacks occurring whether before conducting a security assessment. Or happened later All responsibility shall be sole with the **service user**.

Security Assessment Is Not Financial/Investment Advice Any loss arising from any investment in any project is the responsibility of the investor.

SCRL disclaims any liability incurred. Whether it's Rugpull, Abandonment, Soft Rugpull, Exploit, Exit Scam.

Security Assessment Procedure

1. **Request** The client must submit a formal request and follow the procedure. By submitting the source code and agreeing to the terms of service.
2. **Audit Process** Check for vulnerabilities and vulnerabilities from source code obtained by experts using formal verification methods, including using powerful tools such as Static Analysis, SWC Registry, Dynamic Security Analysis, Automated Security Tools, CWE, Syntax & Parameter Check with AI ,WAS (Warning Avoidance System a python script tools powered by SCRL).
3. **Security Assessment** Deliver Preliminary Security Assessment to clients to acknowledge the risks and vulnerabilities.
4. **Consulting** Discuss on risks and vulnerabilities encountered by clients to apply to their source code to mitigate risks.
   a. **Re-assessment** Reassess the security when the client implements the source code improvements and if the client is satisfied with the results of the audit. We will proceed to the next step.
5. **Full Audit Report** SCRL provides clients with official security assessment reports informing them of risks and vulnerabilities. Officially and it is assumed that the client has been informed of all the information.



Risk Rating

Risk rating using this commonly defined: $Risk\ rating = impact * confidence$

Impact The severity and potential impact of an attacker attack

Confidence Ensuring that attackers expose and use this vulnerability

Confidence Impact [Likelihood]	Low	Medium	High
Low	**Very Low**	**Low**	**Medium**
Medium	**Low**	**Medium**	**High**
High	**Medium**	**High**	**Critical**

Severity is a risk assessment It is calculated from the Impact and Confidence values using the following calculation methods,

$Risk\ rating = impact * confidence$

It is categorized into

7 categories severity based

Gas-optimization	Informational	Very Low	Low	Medium	High	Critical

For **Informational** & **Non-class/Optimization/Best-practices will** <u>not be counted</u> as **severity**

Category

Centralization

Centralization Risk is The risk incurred by a sole proprietor, such as the Owner being able to change something without permission

Economics Risk

Economics Risk is Risks that may affect the economic mechanism system, such as the ability to increase Mint token

Logical Issue

Logical Issue is that can cause errors to core processing, such as any prior operations that cause background processes to crash.

Authorization

Authorization is Possible pitfalls from weak coding allows unrelated people to take any action to modify the values.

Mathematical

Mathematical Any erroneous arithmetic operations affect the operation of the system or lead to erroneous values.

Naming Conventions

Naming Conventions naming variables that may affect code understanding or naming inconsistencies

Security Risk

Security Risk of loss or damage if it's no mitigate

Coding Style

Coding Style is Tips coding for efficiency performance

Best Practices

Best Practices is suggestions for improvement

Optimization

Optimization is performance improvement

Gas Optimization

Gas Optimization is increase performance to avoid expensive gas

Dead Code

Dead Code having unused code This may result in wasted resources and gas fees.

Table Of Content

Summary

- Executive Summary
- CVSS Scoring
- Vulnerability Summary
- Audit Scope
- Audit Version History
- Audit Information
- Smart Contract Audit Summary
- Security Assessment Author
- Digital Sign
- Disclaimer
- Security Assessment Procedure
- Risk Rating
- Category

Source Code Detail

- Dependencies / External Imports
- Visibility, Mutability, Modifier function testing

Vulnerability Finding

- Vulnerability
- SWC Findings
- Contract Description
- Inheritance Relational Graph
- UML Diagram

About SCRL

Source Units Analyzed: **1**
Source Units in Scope: **1** (**100%**)

Type	File	Logic Contracts	Interfaces	Lines	nLines	nSLOC	Comment Lines	Complex. Score	Capabilities
📝 🔍	contracts/DBLUStaking.sol	1	1	274	274	215	3	169	
📝 🔍	**Totals**	**1**	**1**	**274**	**274**	**215**	**3**	**169**	

Legend: [➖]

- **Lines**: total lines of the source unit
- **nLines**: normalized lines of the source unit (e.g. normalizes functions spanning multiple lines)
- **nSLOC**: normalized source lines of code (only source-code lines; no comments, no blank lines)
- **Comment Lines**: lines containing single or block comments
- **Complexity Score**: a custom complexity score derived from code statements that are known to introduce code complexity (branches, loops, calls, external interfaces, ...)

Visibility, Mutability, Modifier function testing

Components

📝Contracts	📚Libraries	🔍Interfaces	🎨Abstract
1	0	1	0

Exposed Functions

This section lists functions that are explicitly declared public or payable. Please note that getter methods for public stateVars are not included.

🌐Public	💰Payable
14	0

External	Internal	Private	Pure	View
11	8	0	1	6

StateVariables

Total	🌐Public
11	7

Capabilities

Solidity Versions observed	🧪 Experimental Features	💰 Can Receive Funds	🖥 Uses Assembly	💣 Has Destroyable Contracts
^0.8.20				

📤 Transfers ETH	⚡ Low-Level Calls	👥 DelegateCall	🧮 Uses Hash Functions	🔖 ECRecover	🌀 New/Create/Create2

♻️ TryCatch	Σ Unchecked

Dependencies / External Imports

Dependency / Import Path	Count
@openzeppelin/contracts/access/Ownable.sol	1
@openzeppelin/contracts/security/ReentrancyGuard.sol	1
@openzeppelin/contracts/token/ERC20/IERC20.sol	1
@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol	1
@openzeppelin/contracts/utils/Counters.sol	1

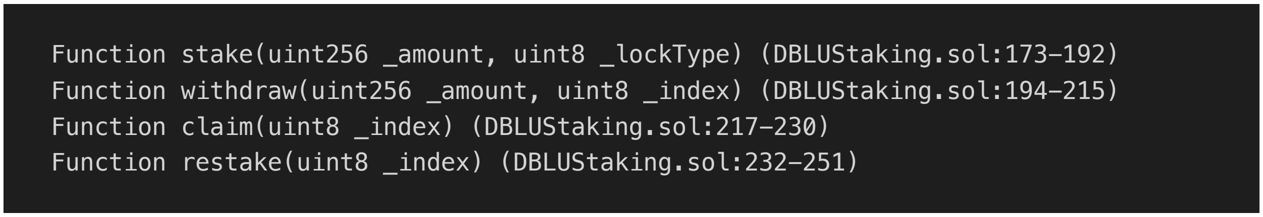
Vulnerability Findings

ID	Vulnerability Detail	Severity	Category	Status
REG-01	Potential Reentrancy Attack	High	Logical Issue	**Resolved**
CEN-01	Centralization Risk	Medium	Centralization	**Mitigate**
SEC-01	Incorrect ERC20 interfaces (erc20-interface)	Medium	Logical Issue	**Resolved**
SEC-02	Unused return values (unused-return)	Medium	Best Practices	**Resolved**
SEC-03	Missing Zero Address Validation (missing-zero-check)	Low	Best Practices	**Resolved**
OPN-01	Unsafe ERC20 operation(s)	Low	Best Practices	**Resolved**
SEC-04	Conformity to Solidity naming conventions	Informational	Naming Conventions	**Resolved**
SEC-05	Detects functions with high (> 11) cyclomatic complexity (cyclomatic-complexity)	Informational	Best Practices	**Acknowledge**
SEC-06	\`require()\` / \`revert()\` statements should have descriptive reason strings	Informational	Coding Style	**Resolved**
SEC-07	Event is missing \`indexed\` fields	Informational	Coding Style	**Acknowledge**
GAS-01	Cache array length outside of loop	Gas-optimization	Gas Optimization	**Resolved**
GAS-02	Use Custom Errors	Gas-optimization	Gas Optimization	**Acknowledge**
GAS-03	Don't initialize variables with default value	Gas-optimization	Gas Optimization	**Acknowledge**
GAS-04	Long revert strings	Gas-optimization	Gas Optimization	**Acknowledge**
GAS-05	Functions guaranteed to revert when called by normal users can be marked \`payable\`	Gas-optimization	Gas Optimization	**Acknowledge**
GAS-06	\`++i\` costs less gas than \`i++\`, especially when it's used in \`for\`-loops (\`--i\`/\`i--\` too)	Gas-optimization	Gas Optimization	**Resolved**
GAS-07	Splitting require() statements that use && saves gas	Gas-optimization	Gas Optimization	**Resolved**
GAS-08	Use != 0 instead of > 0 for unsigned integer comparison	Gas-optimization	Gas Optimization	**Resolved**

REG-01: Potential Reentrancy Attack

Vulnerability Detail	Severity	Location	Category	Status
Potential Reentrancy Attack	High	Check on finding	Logical Issue	**Resolved**

Finding:



```
Function stake(uint256 _amount, uint8 _lockType) (DBLUStaking.sol:173-192)
Function withdraw(uint256 _amount, uint8 _index) (DBLUStaking.sol:194-215)
Function claim(uint8 _index) (DBLUStaking.sol:217-230)
Function restake(uint8 _index) (DBLUStaking.sol:232-251)
```

Description:

Both the 'withdraw' and 'claim' functions involve transferring tokens before updating the state.
This is risky because an external call is made before the state is updated, potentially allowing reentrancy attacks. And we recommend using a reentrancy guard for the 'stake' and 'restake' functions as well.

Recommendation:

Use the Checks-Effects-Interactions pattern and consider adding a reentrancy guard (nonReentrant from OpenZeppelin's ReentrancyGuard).

References: SWC-107: Reentrancy: https://swcregistry.io/docs/SWC-107
OpenZeppelin ReentrancyGuard:
https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard
OpenZeppelin's ReentrancyGuard
https://docs.soliditylang.org/en/v0.7.6/security-considerations.html#re-entrancy

Alleviation:

The doubleup team has already resolved this issue.

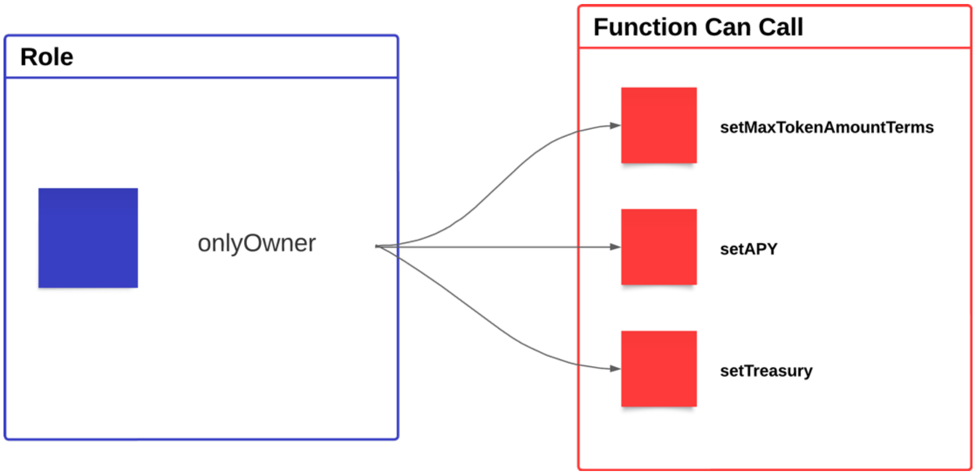
CEN-01: Centralization Risk

Vulnerability Detail	Severity	Location	Category	Status
Centralization Risk	Medium	Check on finding	Centralization	**Mitigate**

Finding:

```
File: DBLUStaking.sol

13: contract DBLUStaking is Ownable {

69:     function setMaxTokenAmountTerms(uint256 amount, uint256 index) external
onlyOwner {

75:     function setAPY(uint8 _apy, uint256 index) external onlyOwner {

85:     function setTreasury(address _treasury) external onlyOwner {

```
```

Explain Function Capability:

The contract provides several functions:

1. **setMaxTokenAmountTerms**
   - This function allows the owner to set the maximum token amount for a specific staking term.

2. **setAPY**
   - enables the owner to adjust the APY for different staking terms.

3. **setTreasury**
   - allows the owner to set or update the treasury address where tokens are managed.
```

Centralization Risk



Recommendation:

In terms of timeframes, there are three categories: short-term, long-term, and permanent.

For short-term solutions, a combination of timelock and multi-signature (2/3 or 3/5) can be used to mitigate risk by delaying sensitive operations and avoiding a single point of failure in key management. This includes implementing a timelock with a reasonable latency, such as 48 hours, for privileged operations; assigning privileged roles to multi-signature wallets to prevent private key compromise; and sharing the timelock contract and multi-signer addresses with the public via a medium/blog link.

For long-term solutions, a combination of timelock and DAO can be used to apply decentralization and transparency to the system. This includes implementing a timelock with a reasonable latency, such as 48 hours, for privileged operations; introducing a DAO/governance/voting module to increase transparency and user involvement; and sharing the timelock contract, multi-signer addresses, and DAO information with the public via a medium/blog link.

Finally, permanent solutions should be implemented to ensure the ongoing security and protection of the system.

Alleviation:

The doubleup team will using multi-signature it’s will mitigated this centralization risk, but still remember **doubleup team still can call this centralized function.**

SEC-01: Incorrect ERC20 interfaces (erc20-interface)

Vulnerability Detail	Severity	Location	Category	Status
Incorrect ERC20 interfaces (erc20-interface)	Medium	Check on finding	Logical Issue	**Resolved**

Finding:

❌ Treasury (src/Treasury.test.sol:7-17) has incorrect ERC20 function interface:Treasury.approve(address,uint256) (src/Treasury.test.sol#14-16)

Recommendation:

Set the appropriate return values and types for the defined `ERC20` functions.

Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-erc20-interface

Alleviation:

The doubleup team has already resolved this issue.

SEC-02: Unused return values (unused-return)

Vulnerability Detail	Severity	Location	Category	Status
Unused return values (unused-return)	Medium	Check on finding	Best Practices	**Resolved**

Finding:

❌ Treasury.approve(address,uint256) (src/Treasury.test.sol:14-16) ignores return value by IERC20(_token).approve(staking,_amount) (src/Treasury.test.sol#15)

Recommendation:

Ensure that all the return values of the function calls are used.

Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return

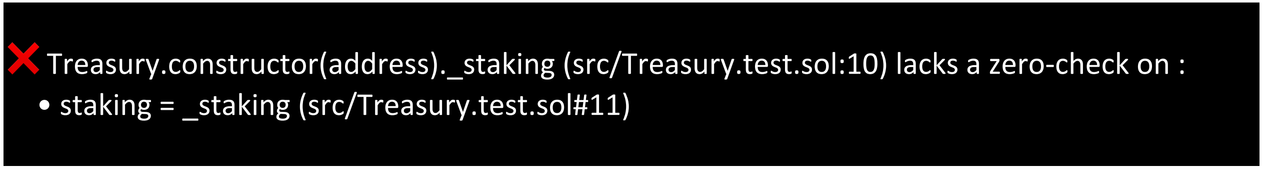
Alleviation:

The doubleup team has already resolved this issue.

SEC-03: Missing Zero Address Validation (missing-zero-check)

Vulnerability Detail	Severity	Location	Category	Status
Missing Zero Address Validation (missing-zero-check)	Low	Check on finding	Best Practices	**Resolved**

Finding:



```
❌ Treasury.constructor(address)._staking (src/Treasury.test.sol:10) lacks a zero-check on :
  • staking = _staking (src/Treasury.test.sol#11)
```

Recommendation:

Check that the address is not zero.

Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation

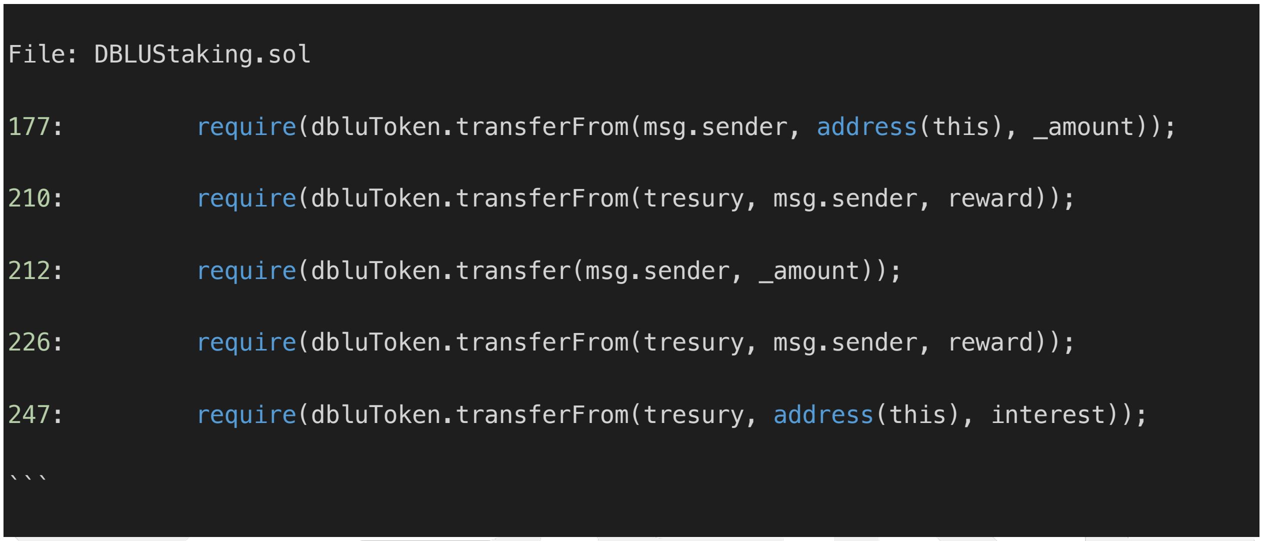
Alleviation:

The doubleup team has already resolved this issue.

OPN-01: Unsafe ERC20 operation(s)

Vulnerability Detail	Severity	Location	Category	Status
Unsafe ERC20 operation(s)	Low	Check on finding	Best Practices	**Resolved**

Finding:

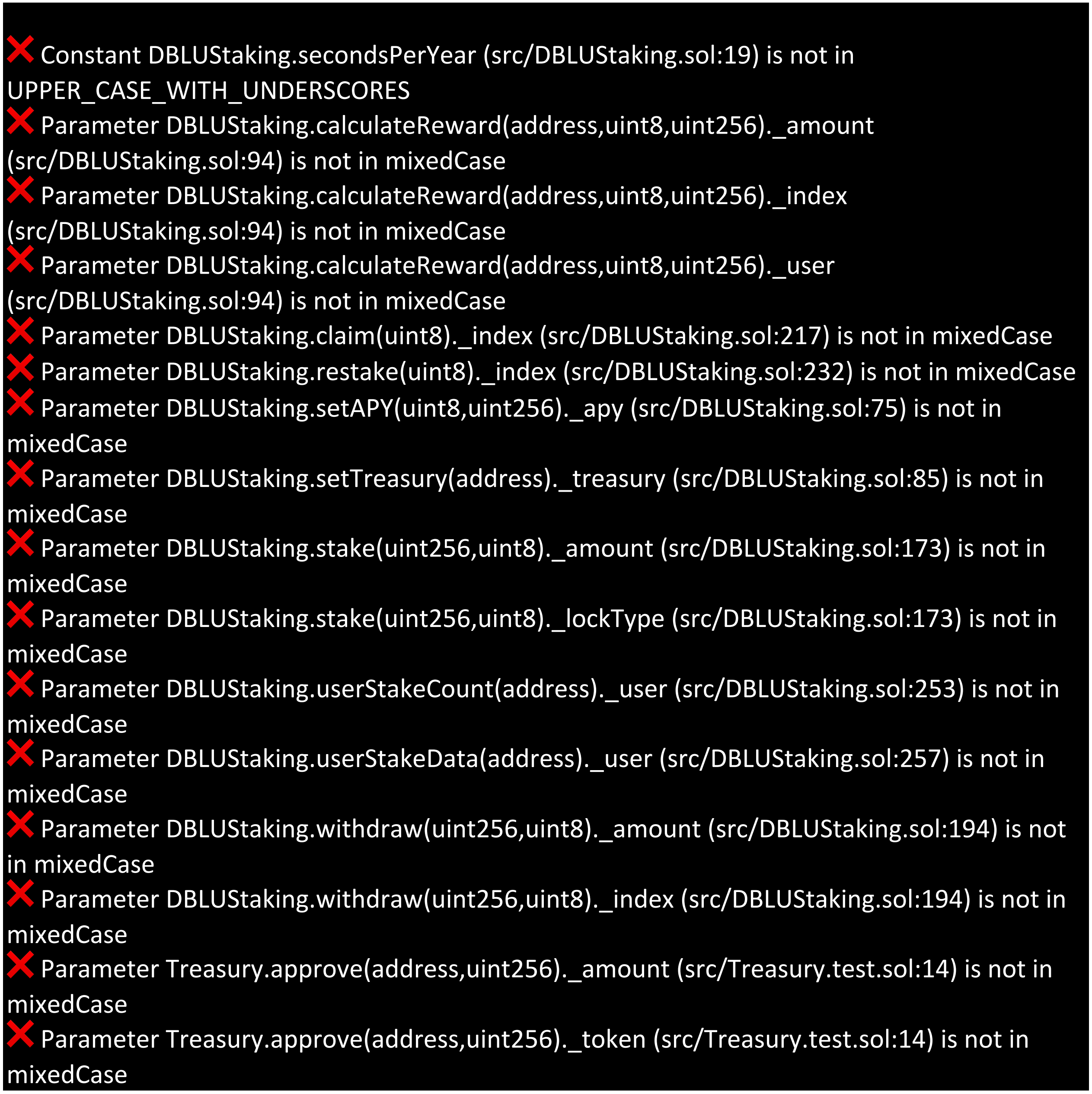


```
File: DBLUStaking.sol

177:        require(dbluToken.transferFrom(msg.sender, address(this), _amount));

210:        require(dbluToken.transferFrom(tresury, msg.sender, reward));

212:        require(dbluToken.transfer(msg.sender, _amount));

226:        require(dbluToken.transferFrom(tresury, msg.sender, reward));

247:        require(dbluToken.transferFrom(tresury, address(this), interest));

```
```

Recommendation:

To mitigate this issue, it is recommended to use OpenZeppelin's SafeERC20 library, which wraps these operations and automatically handles the return value, reverting the transaction if the transfer fails. This approach aligns with the best practices for safe ERC20 interactions as outlined in the OpenZeppelin documentation.

References: CWE-252: Unchecked Return Value: https://cwe.mitre.org/data/definitions/252.html
SWC-104: Unchecked Return Value from Low-Level Calls: https://swcregistry.io/docs/SWC-104
OpenZeppelin SafeERC20 Library:
https://docs.openzeppelin.com/contracts/4.x/api/token/erc20#SafeERC20

Alleviation:

The doubleup team has already resolved this issue.
```

SEC-04: Conformity to Solidity naming conventions (naming-convention)

Vulnerability Detail	Severity	Location	Category	Status
Conformity to Solidity naming conventions (naming-convention)	Informational	Check on finding	Naming Conventions	**Resolved**

Finding:



```
❌ Constant DBLUStaking.secondsPerYear (src/DBLUStaking.sol:19) is not in UPPER_CASE_WITH_UNDERSCORES
❌ Parameter DBLUStaking.calculateReward(address,uint8,uint256)._amount (src/DBLUStaking.sol:94) is not in mixedCase
❌ Parameter DBLUStaking.calculateReward(address,uint8,uint256)._index (src/DBLUStaking.sol:94) is not in mixedCase
❌ Parameter DBLUStaking.calculateReward(address,uint8,uint256)._user (src/DBLUStaking.sol:94) is not in mixedCase
❌ Parameter DBLUStaking.claim(uint8)._index (src/DBLUStaking.sol:217) is not in mixedCase
❌ Parameter DBLUStaking.restake(uint8)._index (src/DBLUStaking.sol:232) is not in mixedCase
❌ Parameter DBLUStaking.setAPY(uint8,uint256)._apy (src/DBLUStaking.sol:75) is not in mixedCase
❌ Parameter DBLUStaking.setTreasury(address)._treasury (src/DBLUStaking.sol:85) is not in mixedCase
❌ Parameter DBLUStaking.stake(uint256,uint8)._amount (src/DBLUStaking.sol:173) is not in mixedCase
❌ Parameter DBLUStaking.stake(uint256,uint8)._lockType (src/DBLUStaking.sol:173) is not in mixedCase
❌ Parameter DBLUStaking.userStakeCount(address)._user (src/DBLUStaking.sol:253) is not in mixedCase
❌ Parameter DBLUStaking.userStakeData(address)._user (src/DBLUStaking.sol:257) is not in mixedCase
❌ Parameter DBLUStaking.withdraw(uint256,uint8)._amount (src/DBLUStaking.sol:194) is not in mixedCase
❌ Parameter DBLUStaking.withdraw(uint256,uint8)._index (src/DBLUStaking.sol:194) is not in mixedCase
❌ Parameter Treasury.approve(address,uint256)._amount (src/Treasury.test.sol:14) is not in mixedCase
❌ Parameter Treasury.approve(address,uint256)._token (src/Treasury.test.sol:14) is not in mixedCase
```

❌ Variable DBLUStaking.APY (src/DBLUStaking.sol:47) is not in mixedCase

Recommendation:
Follow the Solidity [naming convention](https://solidity.readthedocs.io/en/v0.4.25/style-guide.html#naming-conventions).

Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions

Alleviation:
The doubleup team has already resolved this issue.

SEC-05: Detects functions with high (> 11) cyclomatic complexity (cyclomatic-complexity)

Vulnerability Detail	Severity	Location	Category	Status
Detects functions with high (> 11) cyclomatic complexity (cyclomatic-complexity)	Informational	Check on finding	Best Practices	**Acknowledge**

Finding:

❌ DBLUStaking.calculateReward(address,uint8,uint256) (src/DBLUStaking.sol:94-157) has a high cyclomatic complexity (13).

Recommendation:

Reduce cyclomatic complexity by splitting the function into several smaller subroutines.

Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#cyclomatic-complexity

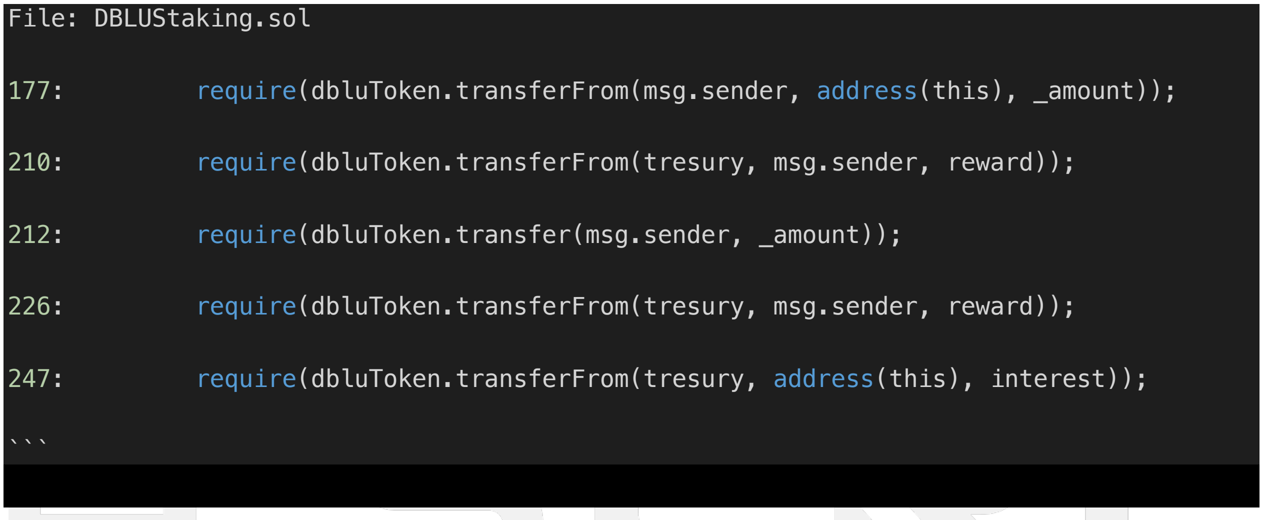
Alleviation:

-

SEC-06: `require()` / `revert()` statements should have descriptive reason strings

Vulnerability Detail	Severity	Location	Category	Status
`require()` / `revert()` statements should have descriptive reason strings	Informational	Check on finding	Coding Style	**Resolved**

Finding:



```
File: DBLUStaking.sol

177:        require(dbluToken.transferFrom(msg.sender, address(this), _amount));

210:        require(dbluToken.transferFrom(tresury, msg.sender, reward));

212:        require(dbluToken.transfer(msg.sender, _amount));

226:        require(dbluToken.transferFrom(tresury, msg.sender, reward));

247:        require(dbluToken.transferFrom(tresury, address(this), interest));

```
```

Recommendation:

The require() statements in the DBLUStaking contract lack descriptive reason strings. Including descriptive reason strings in require() and revert() statements helps in identifying the exact reason for failure, making debugging easier and improving the overall readability and maintainability of the code.

Add descriptive reason strings to all require() statements to clearly indicate the reason for the failure.

Reference: Solidity documentation on error handling.
https://docs.soliditylang.org/en/v0.7.6/control-structures.html#error-handling-assert-require-revert-and-exceptions

Alleviation:

The doubleup team has already resolved this issue.
```

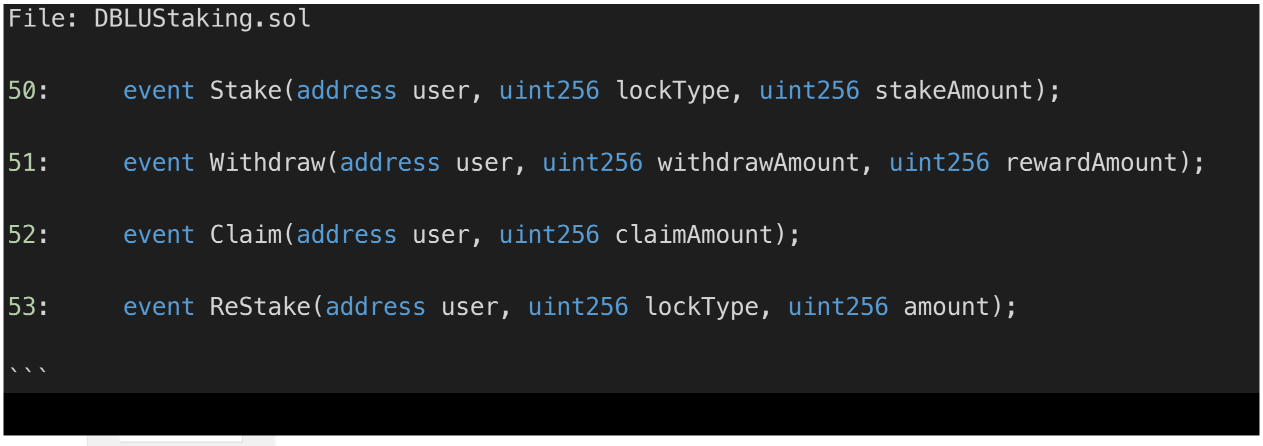
SEC-07: Event is missing `indexed` fields

Vulnerability Detail	Severity	Location	Category	Status
Event is missing `indexed` fields	Informational	Check on finding	Coding Style	Acknowledge

Impact:

Without indexed fields, querying these events from off-chain tools becomes less efficient, potentially leading to higher costs and slower performance for applications that rely on these events.

Finding:

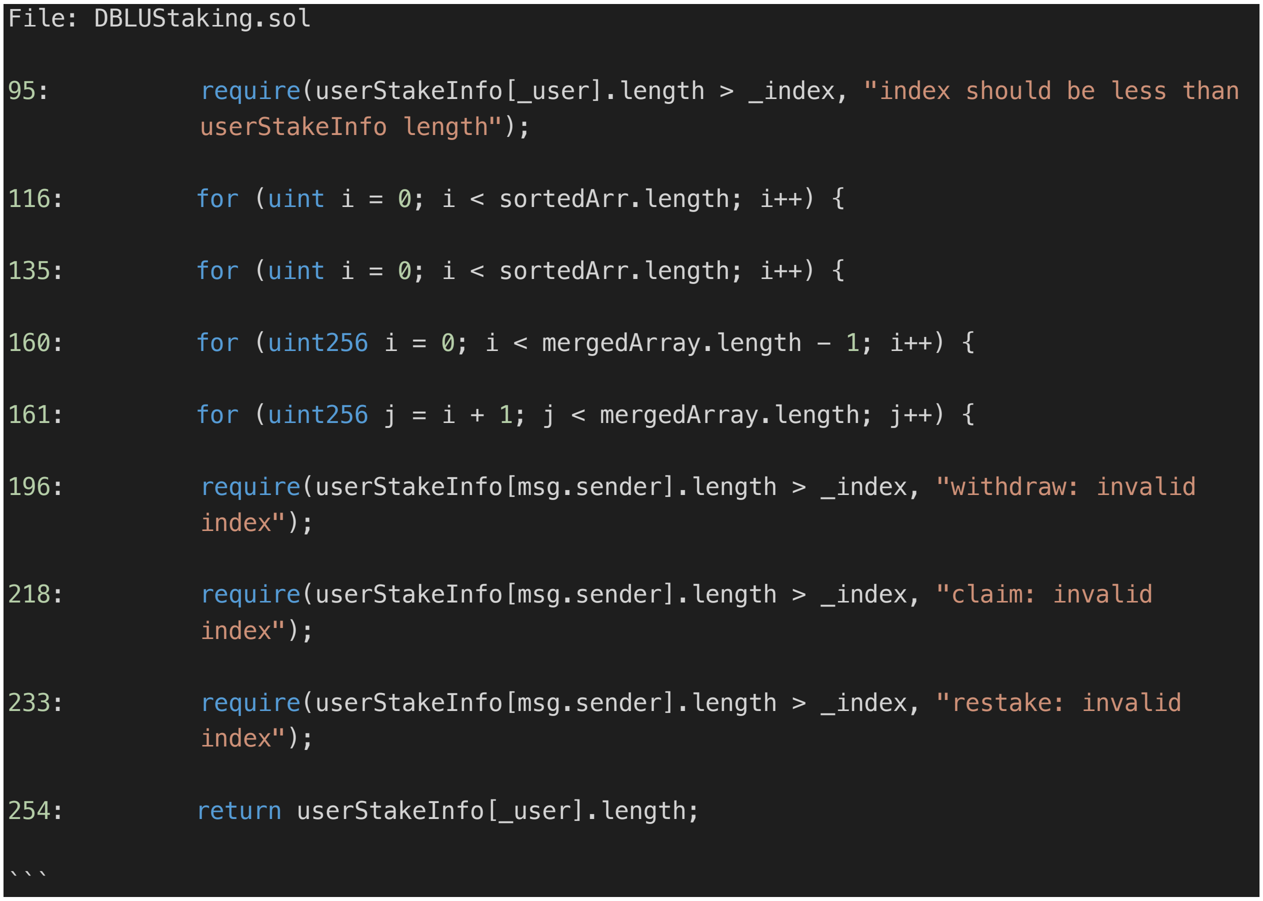


```
File: DBLUStaking.sol

50:      event Stake(address user, uint256 lockType, uint256 stakeAmount);

51:      event Withdraw(address user, uint256 withdrawAmount, uint256 rewardAmount);

52:      event Claim(address user, uint256 claimAmount);

53:      event ReStake(address user, uint256 lockType, uint256 amount);

```
```

Recommendation:

The events in the DBLUStaking contract are missing indexed fields. Indexing event fields makes them more quickly accessible to off-chain tools that parse events. Each event should use indexed fields where appropriate to facilitate efficient querying and filtering.

Add indexed keywords to relevant fields in the events. Typically, fields that are used as filters in queries should be indexed. In this case, the user address should be indexed, as well as other fields that might be commonly used for filtering.

Reference: Solidity documentation on error handling.
https://docs.soliditylang.org/en/v0.7.6/contracts.html#events

Alleviation:

-
```

GAS-01: Cache array length outside of loop

Vulnerability Detail	Severity	Location	Category	Status
Cache array length outside of loop	-	Check on finding	Gas Optimization	**Resolved**

Finding:

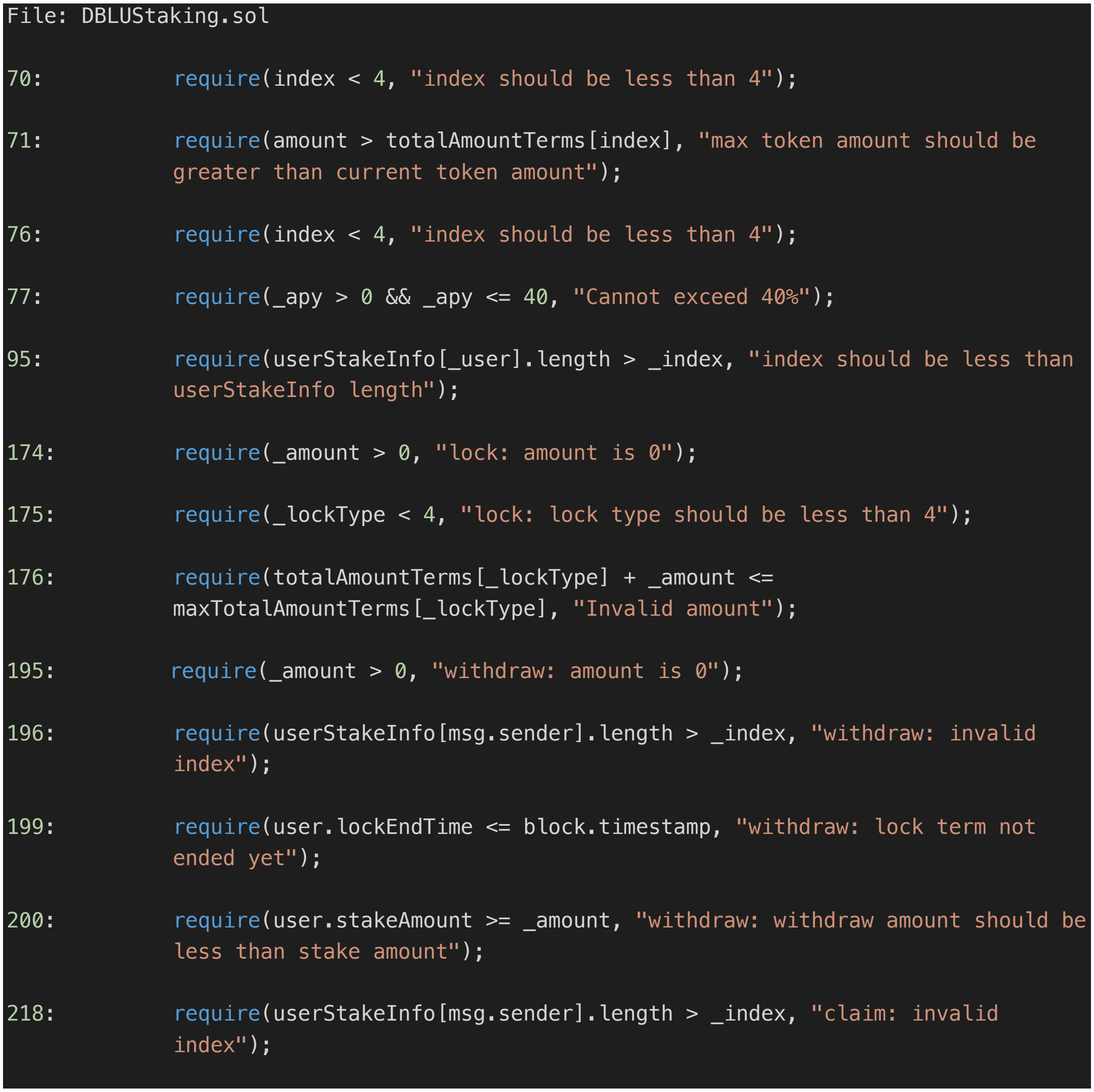


```
File: DBLUStaking.sol

95:          require(userStakeInfo[_user].length > _index, "index should be less than
             userStakeInfo length");

116:         for (uint i = 0; i < sortedArr.length; i++) {

135:         for (uint i = 0; i < sortedArr.length; i++) {

160:         for (uint256 i = 0; i < mergedArray.length - 1; i++) {

161:         for (uint256 j = i + 1; j < mergedArray.length; j++) {

196:         require(userStakeInfo[msg.sender].length > _index, "withdraw: invalid
             index");

218:         require(userStakeInfo[msg.sender].length > _index, "claim: invalid
             index");

233:         require(userStakeInfo[msg.sender].length > _index, "restake: invalid
             index");

254:         return userStakeInfo[_user].length;

```
```

Recommendation:

If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

Alleviation:

The doubleup team has already resolved this issue.
```

GAS-02: Use Custom Errors

Vulnerability Detail	Severity	Location	Category	Status
Use Custom Errors	-	Check on finding	Gas Optimization	Acknowledge

Finding:



```
File: DBLUStaking.sol

70:           require(index < 4, "index should be less than 4");

71:           require(amount > totalAmountTerms[index], "max token amount should be
              greater than current token amount");

76:           require(index < 4, "index should be less than 4");

77:           require(_apy > 0 && _apy <= 40, "Cannot exceed 40%");

95:           require(userStakeInfo[_user].length > _index, "index should be less than
              userStakeInfo length");

174:          require(_amount > 0, "lock: amount is 0");

175:          require(_lockType < 4, "lock: lock type should be less than 4");

176:          require(totalAmountTerms[_lockType] + _amount <=
              maxTotalAmountTerms[_lockType], "Invalid amount");

195:          require(_amount > 0, "withdraw: amount is 0");

196:          require(userStakeInfo[msg.sender].length > _index, "withdraw: invalid
              index");

199:          require(user.lockEndTime <= block.timestamp, "withdraw: lock term not
              ended yet");

200:          require(user.stakeAmount >= _amount, "withdraw: withdraw amount should be
              less than stake amount");

218:          require(userStakeInfo[msg.sender].length > _index, "claim: invalid
              index");
```

~~~
233:          require(userStakeInfo[msg.sender].length > _index, "restake: invalid
              index");

236:          require(user.lockEndTime <= block.timestamp, "restake: lock term not
              ended yet");

237:          require(user.stakeAmount > 0, "restake: staked amount is 0");

```
~~~

Recommendation:

[Source](https://blog.soliditylang.org/2021/04/21/custom-errors/)
Instead of using error strings, to reduce deployment and runtime cost, you should use Custom Errors. This would save both deployment and runtime cost.

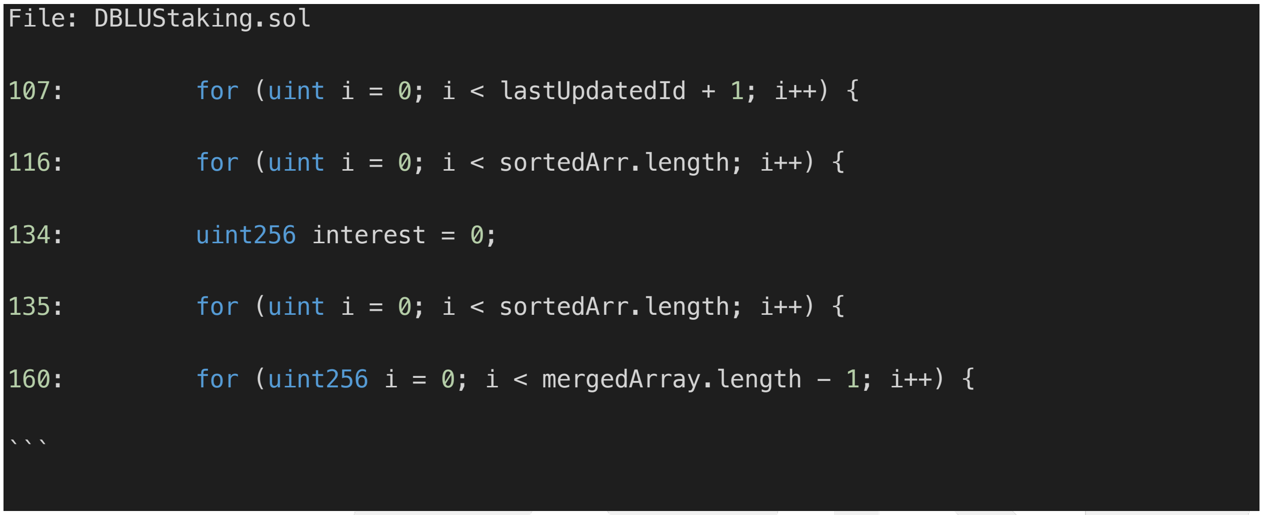
Alleviation:

-

GAS-03: Don't initialize variables with default value

Vulnerability Detail	Severity	Location	Category	Status
Don't initialize variables with default value	-	Check on finding	Gas Optimization	**Acknowledge**

Finding:



```
File: DBLUStaking.sol

107:          for (uint i = 0; i < lastUpdatedId + 1; i++) {

116:          for (uint i = 0; i < sortedArr.length; i++) {

134:          uint256 interest = 0;

135:          for (uint i = 0; i < sortedArr.length; i++) {

160:          for (uint256 i = 0; i < mergedArray.length - 1; i++) {

```
```

Recommendation:

In Solidity, it is unnecessary to initialize variables with their default values since Solidity automatically sets storage variables to their default value. Initializing variables with default values can lead to redundant code, making the codebase less clean and potentially confusing.

Remove unnecessary initializations where the variables are set to their default values.

Reference: Solidity documentation
https://docs.soliditylang.org/en/v0.7.6/style-guide.html

Alleviation:

-
```

# GAS-04: Long revert strings

| Vulnerability Detail | Severity | Location | Category | Status |
|---|---|---|---|---|
| Long revert strings | Gas-optimization | Check on finding | Gas Optimization | **Acknowledge** |

## Finding:

```
File: DBLUStaking.sol

71:           require(amount > totalAmountTerms[index], "max token amount should be
              greater than current token amount");

95:           require(userStakeInfo[_user].length > _index, "index should be less than
              userStakeInfo length");

175:          require(_lockType < 4, "lock: lock type should be less than 4");

199:          require(user.lockEndTime <= block.timestamp, "withdraw: lock term not
              ended yet");

200:          require(user.stakeAmount >= _amount, "withdraw: withdraw amount should be
              less than stake amount");

```
```

Recommendation:

Long revert strings in the require statements consume more gas. It’s more gas-efficient to use short revert strings or error codes. In addition, using custom errors can further optimize gas usage while providing clear and meaningful error messages.

Alleviation:

-
```

# GAS-05: Functions guaranteed to revert when called by normal users can be marked `payable`

| Vulnerability Detail | Severity | Location | Category | Status |
|---|---|---|---|---|
| Functions guaranteed to revert when called by normal users can be marked `payable` | Gas-optimization | Check on finding | Gas Optimization | **Acknowledge** |

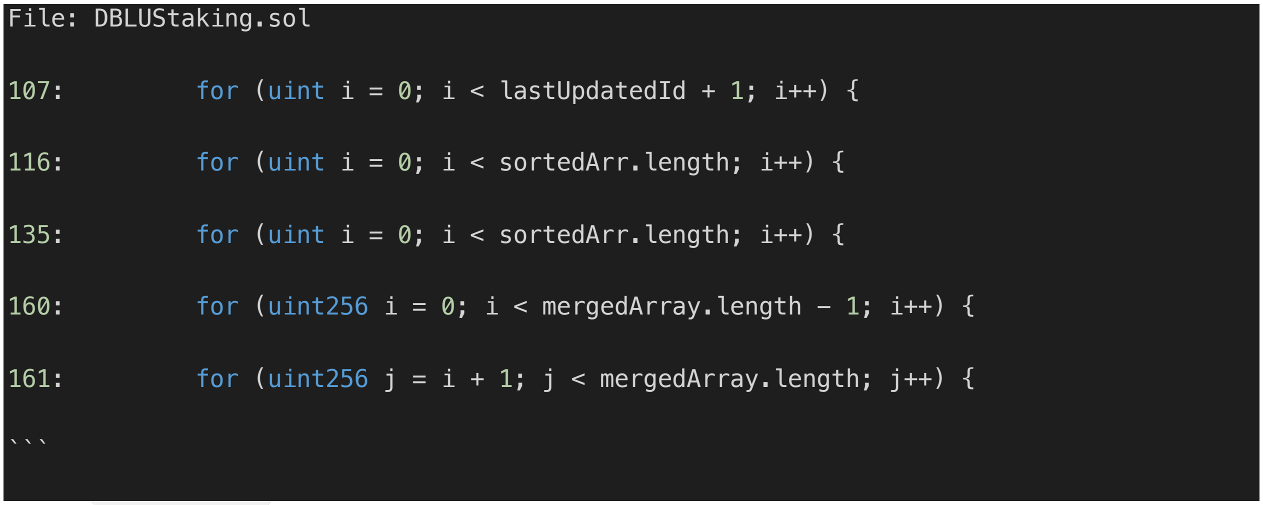
## Finding:

```
File: DBLUStaking.sol

69:     function setMaxTokenAmountTerms(uint256 amount, uint256 index) external
        onlyOwner {

75:     function setAPY(uint8 _apy, uint256 index) external onlyOwner {

85:     function setTreasury(address _treasury) external onlyOwner {

```
```

Recommendation:

Functions that are restricted to certain roles (e.g., onlyOwner) and will revert if called by normal users can be marked as payable. This reduces gas costs for legitimate callers by eliminating the need for the compiler to include checks for whether a payment was provided.

Mark the setMaxTokenAmountTerms, setAPY, and setTreasury functions as payable.

Reference: Solidity documentation on function modifiers and payable
https://docs.soliditylang.org/en/v0.7.6/contracts.html#function-modifiers

Alleviation:

-
```

# GAS-06: `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)

| Vulnerability Detail | Severity | Location | Category | Status |
|---|---|---|---|---|
| `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too) | Gas-optimization | Check on finding | Gas Optimization | **Resolved** |

## Finding:



~~~
File: DBLUStaking.sol

107:         for (uint i = 0; i < lastUpdatedId + 1; i++) {

116:         for (uint i = 0; i < sortedArr.length; i++) {

135:         for (uint i = 0; i < sortedArr.length; i++) {

160:         for (uint256 i = 0; i < mergedArray.length - 1; i++) {

161:         for (uint256 j = i + 1; j < mergedArray.length; j++) {

```
~~~

Recommendation:

Using ++i (pre-increment) instead of i++ (post-increment) can save gas, especially in for loops. The same principle applies to decrement operations (--i vs i--).

Change post-increment i++ to pre-increment ++i to optimize gas usage.

Alleviation:

The doubleup team has already resolved this issue.

GAS-07: Splitting require() statements that use && saves gas

Vulnerability Detail	Severity	Location	Category	Status
Splitting require() statements that use && saves gas	Gas-optimization	Check on finding	Gas Optimization	**Resolved**

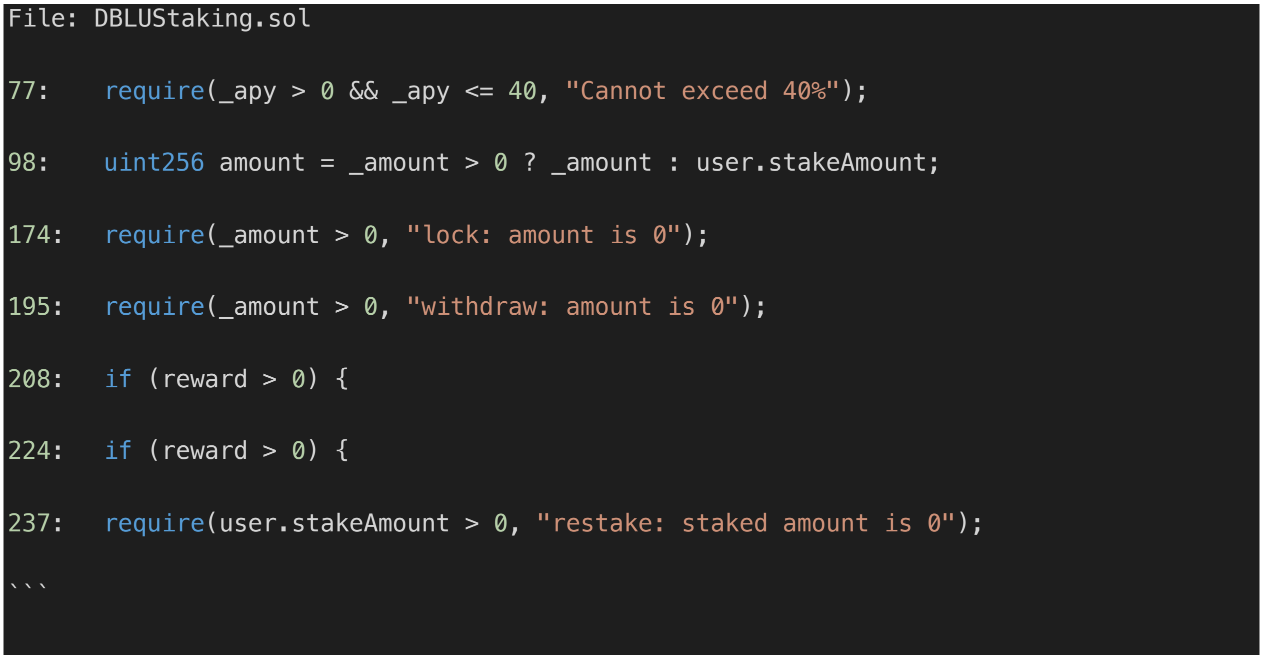
Finding:

```
File: DBLUStaking.sol

77:         require(_apy > 0 && _apy <= 40, "Cannot exceed 40%");

```
```

Recommendation:

Combining multiple conditions in a single require() statement using the **&&** operator can lead to higher gas consumption. Splitting the require() statements into separate checks can save gas because the second condition is only evaluated if the first condition is true.

Split the require() statement into two separate checks.

Alleviation:

The doubleup team has already resolved this issue.
```

GAS-08: Use != 0 instead of > 0 for unsigned integer comparison

Vulnerability Detail	Severity	Location	Category	Status
Use != 0 instead of > 0 for unsigned integer comparison	-	Check on finding	Gas Optimization	**Resolved**

Finding:



```
File: DBLUStaking.sol

77:    require(_apy > 0 && _apy <= 40, "Cannot exceed 40%");

98:    uint256 amount = _amount > 0 ? _amount : user.stakeAmount;

174:   require(_amount > 0, "lock: amount is 0");

195:   require(_amount > 0, "withdraw: amount is 0");

208:   if (reward > 0) {

224:   if (reward > 0) {

237:   require(user.stakeAmount > 0, "restake: staked amount is 0");

```
```

Alleviation:

The doubleup team has already resolved this issue.
```

SWC Findings

ID	Title	Scanning	Result
SWC-100	Function Default Visibility	Complete	No risk
SWC-101	Integer Overflow and Underflow	Complete	No risk
SWC-102	Outdated Compiler Version	Complete	No risk
SWC-103	Floating Pragma	Complete	No risk
SWC-104	Unchecked Call Return Value	Complete	No risk
SWC-105	Unprotected Ether Withdrawal	Complete	No risk
SWC-106	Unprotected SELFDESTRUCT Instruction	Complete	No risk
SWC-107	Reentrancy	Complete	No risk
SWC-108	State Variable Default Visibility	Complete	No risk
SWC-109	Uninitialized Storage Pointer	Complete	No risk
SWC-110	Assert Violation	Complete	No risk
SWC-111	Use of Deprecated Solidity Functions	Complete	No risk
SWC-112	Delegatecall to Untrusted Callee	Complete	No risk
SWC-113	DoS with Failed Call	Complete	No risk
SWC-114	Transaction Order Dependence	Complete	No risk
SWC-115	Authorization through tx.origin	Complete	No risk

SWC-116	Block values as a proxy for time	Complete	No risk
SWC-117	Signature Malleability	Complete	No risk
SWC-118	Incorrect Constructor Name	Complete	No risk
SWC-119	Shadowing State Variables	Complete	No risk
SWC-120	Weak Sources of Randomness from Chain Attributes	Complete	No risk
SWC-121	Missing Protection against Signature Replay Attacks	Complete	No risk
SWC-122	Lack of Proper Signature Verification	Complete	No risk
SWC-123	Requirement Violation	Complete	No risk
SWC-124	Write to Arbitrary Storage Location	Complete	No risk
SWC-125	Incorrect Inheritance Order	Complete	No risk
SWC-126	Insufficient Gas Griefing	Complete	No risk
SWC-127	Arbitrary Jump with Function Type Variable	Complete	No risk
SWC-128	DoS With Block Gas Limit	Complete	No risk
SWC-129	Typographical Error	Complete	No risk
SWC-130	Right-To-Left-Override control character (U+202E)	Complete	No risk
SWC-131	Presence of unused variables	Complete	No risk
SWC-132	Unexpected Ether balance	Complete	No risk

SWC-133	Hash Collisions With Multiple Variable Length Arguments	Complete	No risk
SWC-134	Message call with hardcoded gas amount	Complete	No risk
SWC-135	Code With No Effects	Complete	No risk
SWC-136	Unencrypted Private Data On-Chain	Complete	No risk

Contracts Description Table

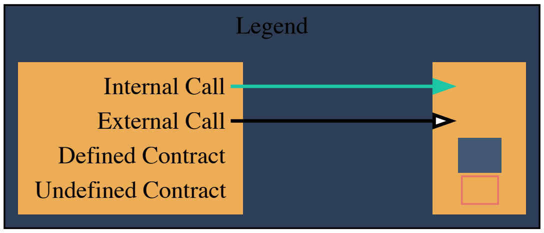
Contract	Type	Bases		
└	**Function Name**	**Visibility**	**Mutability**	**Modifiers**
DBLU	Interface	IERC20		
DBLUStaking	Implementation	Ownable, ReentrancyGuard		
└		Public ❗	🛑	NO❗
└	getAmountTerms	External ❗		NO❗
└	getMaxAmountTerms	External ❗		NO❗
└	setMaxTokenAmountTerms	External ❗	🛑	onlyOwner
└	setAPY	External ❗	🛑	onlyOwner
└	setTreasury	External ❗	🛑	onlyOwner
└	getBlockTime	Public ❗		NO❗
└	calculateReward	Public ❗		NO❗
└	sort	Public ❗		NO❗
└	stake	External ❗	🛑	nonReentrant
└	withdraw	External ❗	🛑	nonReentrant
└	claim	External ❗	🛑	nonReentrant
└	restake	External ❗	🛑	nonReentrant
└	userStakeCount	External ❗		NO❗

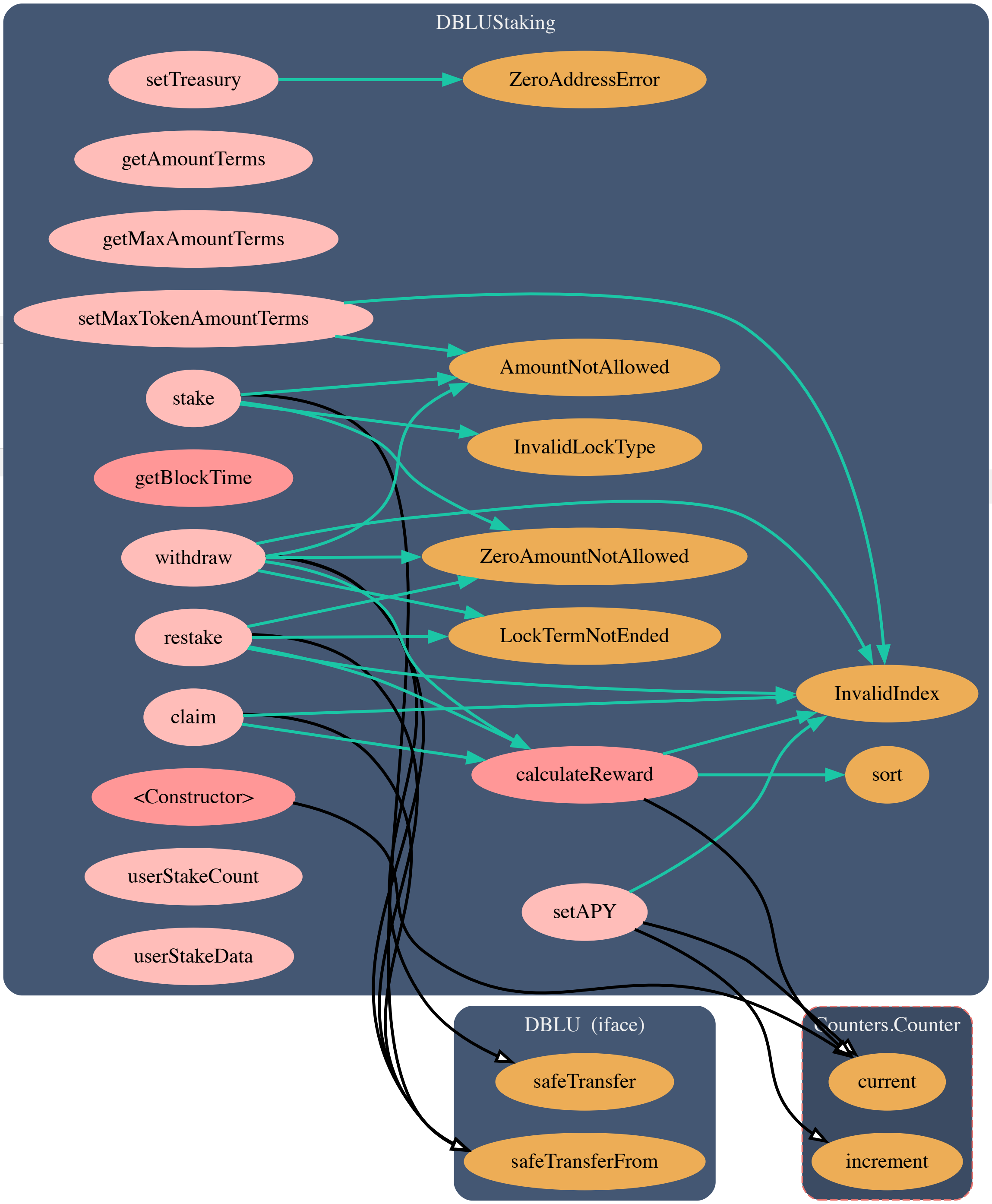
Contract	Type	Bases		
└	userStakeData	External ❗		NO❗

Legend

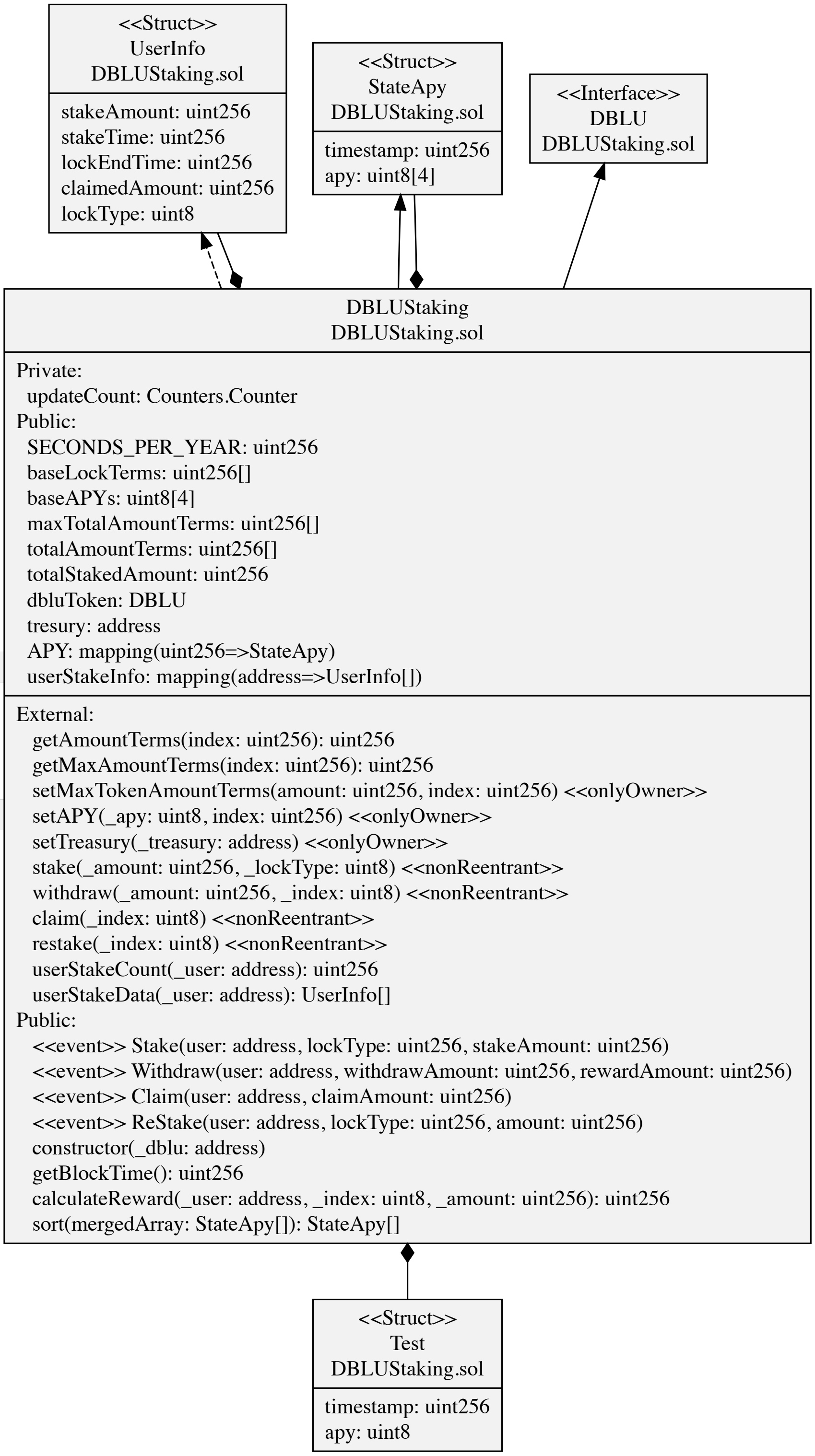
Symbol	Meaning
🛑	Function can modify state
💵	Function is payable

Call Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract

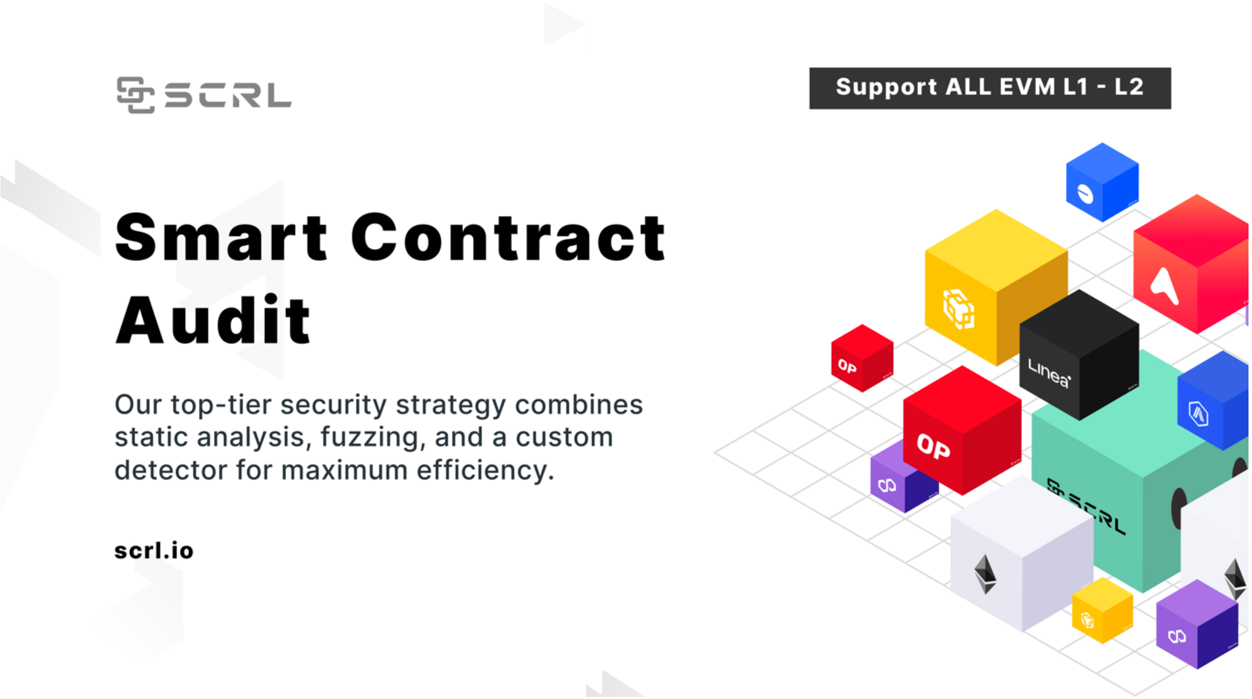

DBLUStaking
setTreasury
ZeroAddressError
getAmountTerms
getMaxAmountTerms
setMaxTokenAmountTerms
AmountNotAllowed
stake
InvalidLockType
getBlockTime
withdraw
ZeroAmountNotAllowed
restake
LockTermNotEnded
InvalidIndex
claim
calculateReward
sort
<Constructor>
userStakeCount
setAPY
userStakeData
DBLU (iface)
safeTransfer
safeTransferFrom
Counters.Counter
current
increment

UML Class Diagram


<<Struct>>
UserInfo
DBLUStaking.sol
stakeAmount: uint256
stakeTime: uint256
lockEndTime: uint256
claimedAmount: uint256
lockType: uint8
<<Struct>>
StateApy
DBLUStaking.sol
timestamp: uint256
apy: uint8[4]
<<Interface>>
DBLU
DBLUStaking.sol
DBLUStaking
DBLUStaking.sol
Private:
updateCount: Counters.Counter
Public:
SECONDS_PER_YEAR: uint256
baseLockTerms: uint256[]
baseAPYs: uint8[4]
maxTotalAmountTerms: uint256[]
totalAmountTerms: uint256[]
totalStakedAmount: uint256
dbluToken: DBLU
tresury: address
APY: mapping(uint256=>StateApy)
userStakeInfo: mapping(address=>UserInfo[])
External:
getAmountTerms(index: uint256): uint256
getMaxAmountTerms(index: uint256): uint256
setMaxTokenAmountTerms(amount: uint256, index: uint256) <<onlyOwner>>
setAPY(_apy: uint8, index: uint256) <<onlyOwner>>
setTreasury(_treasury: address) <<onlyOwner>>
stake(_amount: uint256, _lockType: uint8) <<nonReentrant>>
withdraw(_amount: uint256, _index: uint8) <<nonReentrant>>
claim(_index: uint8) <<nonReentrant>>
restake(_index: uint8) <<nonReentrant>>
userStakeCount(_user: address): uint256
userStakeData(_user: address): UserInfo[]
Public:
<<event>> Stake(user: address, lockType: uint256, stakeAmount: uint256)
<<event>> Withdraw(user: address, withdrawAmount: uint256, rewardAmount: uint256)
<<event>> Claim(user: address, claimAmount: uint256)
<<event>> ReStake(user: address, lockType: uint256, amount: uint256)
constructor(_dblu: address)
getBlockTime(): uint256
calculateReward(_user: address, _index: uint8, _amount: uint256): uint256
sort(mergedArray: StateApy[]): StateApy[]
<<Struct>>
Test
DBLUStaking.sol
timestamp: uint256
apy: uint8

About SCRL

SCRL (Previously name SECURI LAB) was established in 2020, and its goal is to deliver a security solution for Web3 projects by expert security researchers. To verify the security of smart contracts, they have developed internal tools and KYC solutions for Web3 projects using industry-standard technology. SCRL was created to solve security problems for Web3 projects. They focus on technology for conciseness in security auditing. They have developed Python-based tools for their internal use called WAS and SCRL. Their goal is to drive the crypto industry in Thailand to grow with security protection technology.



SCRL

Support ALL EVM L1 - L2

Smart Contract Audit

Our top-tier security strategy combines static analysis, fuzzing, and a custom detector for maximum efficiency.

scrl.io

Follow Us On:

Website	https://scrl.io/
Twitter	https://twitter.com/scrl_io
Telegram	https://t.me/scrl_io
Medium	https://scrl.medium.com/