TECHFUND Audit Score

Smart contract Audit Report

Kiheitai Audit

Completed on 2022-04-20

Score **POSITIVE**

Risk level		
	Critical	0
	High	0
	Medium	1
	Low	0
	Note	2

Risk level detail

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

The tester arrives at the likelihood and impact estimates, they can now combine them to get a final severity rating for this risk. Note that if they have good business impact information, they should use that instead of the technical impact information.

https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

Vulnerability Review

Number of warnings

Compiler Version	1
State Variable Default Visibility	0
In-Correct Estimation of Days from Date	1
Costly Loops for external calls	0
Integer Overflow / Underflow	0
Parity Multisig Bug	0
Callstack Depth Attack	0
Error in setting up Dynamic Arrays	0
Gas wastage because of un-necessary bound checks	1
Re-Entrancy (out of order events)	0
Double Withdrawal	0

TECHFUND Audit Score

Smart contract Audit Report

Compiler Version

1

Solidity version used is inconsistent & not fixed. It is highly recommended to update the Solidity version to the latest. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statements and using a fixed version of Solidity upon understanding the version specifications.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
```

In-Correct Estimation of Days from Date

1

Solidity integer division might truncate. As a result, performing multiplication before division can sometimes avoid loss of precision.
In the case of JavaScript, following issue happens due to floating point errors. In Solidity we don't have floating points, but we get a similar issue and we can get "rounding errors".
For example : In Solidity the second operation would actually yield 2999. By doing all our multiplications first, we mitigate rounding related issues as much as possible. Hence it is suggested to do all multiplication before division otherwise the output can be in-correct.

```
» console.log((30 * 100 * 13) / 13)
3000
← undefined
» console.log((30 / 13) * 100 * 13)
2999.9999999999995
← undefined
»
```

```
function _daysToDate(
    uint256 _days⬆
) internal pure returns (uint256 year⬆, uint256 month⬆, uint256 day⬆) {
    unchecked {
        int256 __days = int256(_days⬆);

        int256 L = __days + 68569 + OFFSET19700101;
        int256 N = (4 * L) / 146097;
        L = L - (146097 * N + 3) / 4;
        int256 _year = (4000 * (L + 1)) / 1461001;
        L = L - (1461 * _year) / 4 + 31;
```

Because of the above issue in _daysToDate, the return will not be correctly calculated. It is highly suggested to do the multiplications first and then proceed to do the division. Otherwise there can be offsets in expected and actual values.

Gas wastage because of un-necessary bound checks

1

When an array is accessed in Solidity, the language automatically performs a check to ensure that the index used to access the array is smaller than the length of the array. This check is known as a bounds check. The purpose of the bounds check is to prevent an out-of-bounds error from occurring during the execution of the smart contract, which could lead to unexpected behaviour.

However, there may be certain situations where a developer can be certain that an index used to access an array will always be within a specific range. In these scenarios, the developer can choose to skip the bounds check to save on gas costs. Skipping the bounds check can reduce the amount of gas required for the transaction, resulting in a lower overall cost for the user.

This is in particular not a security issue, hence has been mentioned as a note.
We would highly suggest to use the implementation from the current OpenZeppelin reference
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/7c5f6bc2c8743d83443fa46395d75f2f3f99054a/contracts/utils/Arrays.sol#L37

```
if (array[mid].blockNumber > blockNumber⬆) {
    high = mid;
} else {
    low = mid + 1;
}
```

findUpperBound (un-optimised)

```
if (unsafeAccess(array, mid).value > element) {
    high = mid;
} else {
    low = mid + 1;
}
```

```
assembly {
    mstore(0, arr.slot)
    slot := add(keccak256(0, 0x20), pos)
}
return slot.getAddressSlot();
```

findUpperBound (optimised)