TECHFUND Audit Score

Smart contract Audit Report

Cyberstella Audit

Completed on 2022-02-14

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

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

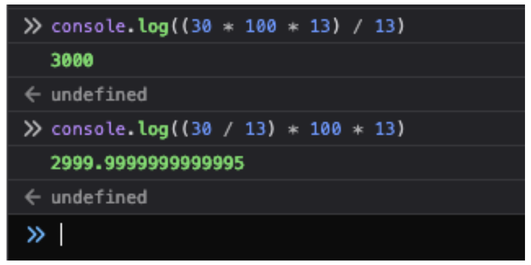
Vulnerability Review

Vulnerability	Number of warnings
In-Correct Estimation of Available amount	1
Costly Loops for external calls	1
Compiler Version	1
State Variable Default Visibility	0
Integer Overflow / Underflow	0
Parity Multisig Bug	0
Callstack Depth Attack	0
Error in setting up Dynamic Arrays	0
In-sufficient transfer check while Minting Tokens	0
Re-Entrancy	0
Double Withdrawal	0

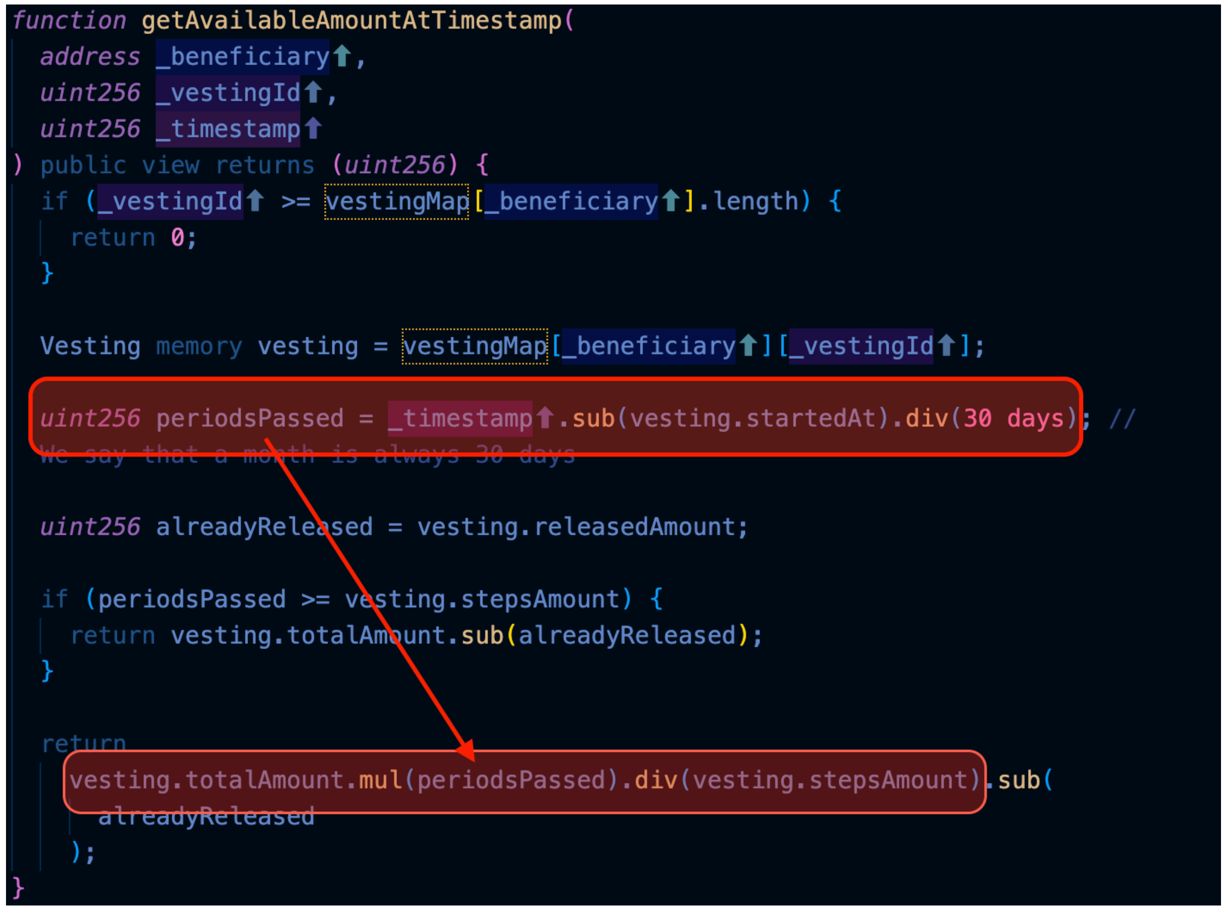
In-Correct Estimation of Available amount

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
>> |
```



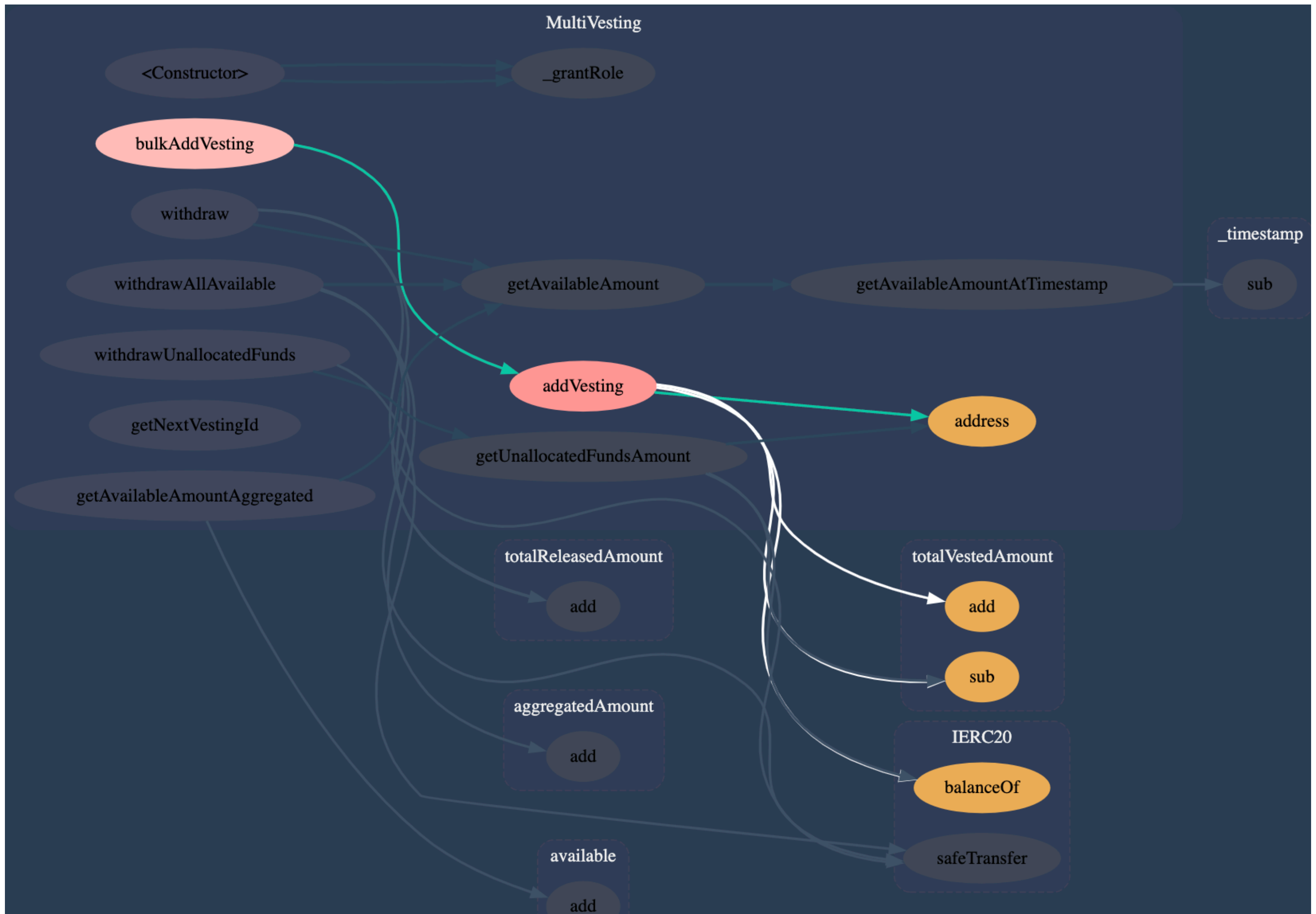
```
function getAvailableAmountAtTimestamp(
  address _beneficiary,
  uint256 _vestingId,
  uint256 _timestamp
) public view returns (uint256) {
  if (_vestingId >= vestingMap[_beneficiary].length) {
    return 0;
  }

  Vesting memory vesting = vestingMap[_beneficiary][_vestingId];

  uint256 periodsPassed = _timestamp.sub(vesting.startedAt).div(30 days); // We say that a month is always 30 days

  uint256 alreadyReleased = vesting.releasedAmount;

  if (periodsPassed >= vesting.stepsAmount) {
    return vesting.totalAmount.sub(alreadyReleased);
  }

  return
    vesting.totalAmount.mul(periodsPassed).div(vesting.stepsAmount).sub(
      alreadyReleased
    );
}
```

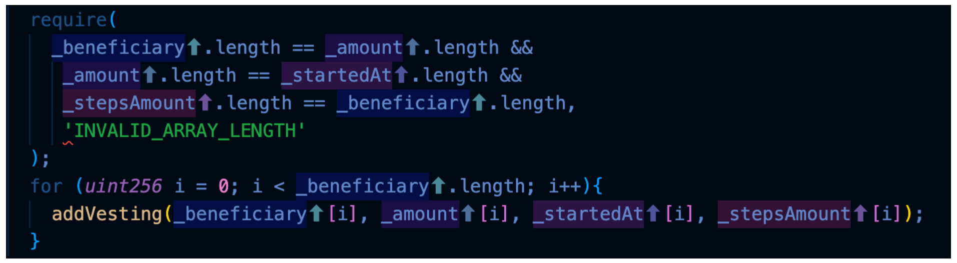
Because of the above issue in MultiVesting, the amount will not be correctly calculated. It is highly suggested to do the multiplications first and then proceed to do the division.

Costly Loops for external calls

1

A peculiar case of excessive gas abuse can be seen in scenarios where there's a parameter that is directly used inside loops without any validation on the number of times for which the loop runs.
The functions should validate that the users can't control the variable length used inside the loop to traverse a large amount of data. If it can't be omitted, then there should be a limit on the length as per the code logic.
Whenever loops are used in Solidity, the developers should pay special attention to the actions happening inside the loop to make sure that the transaction does not consume excessive gas and does not go over the gas limit.
To minimize the damage caused by such failures inside the loops, it is better to isolate each external call into its own transaction that can be initiated by the recipient of the call.





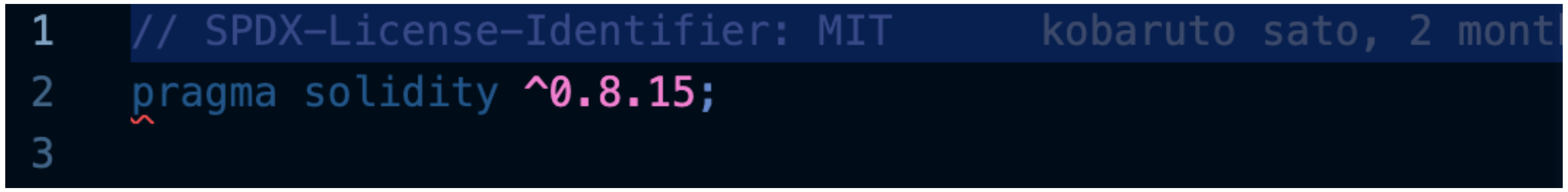
```
require(
  _beneficiary.length == _amount.length &&
   _amount.length == _startedAt.length &&
   _stepsAmount.length == _beneficiary.length,
   'INVALID_ARRAY_LENGTH'
);
for (uint256 i = 0; i < _beneficiary.length; i++){
  addVesting(_beneficiary[i], _amount[i], _startedAt[i], _stepsAmount[i]);
}
```

addBulkVesting has calls inside a loop

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      kobaruto sato, 2 mont
pragma solidity ^0.8.15;

```