TECHFUND Audit Score
Smart contract Audit Report

Astarfarm Audit Report

Completed on 2022-05-17

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

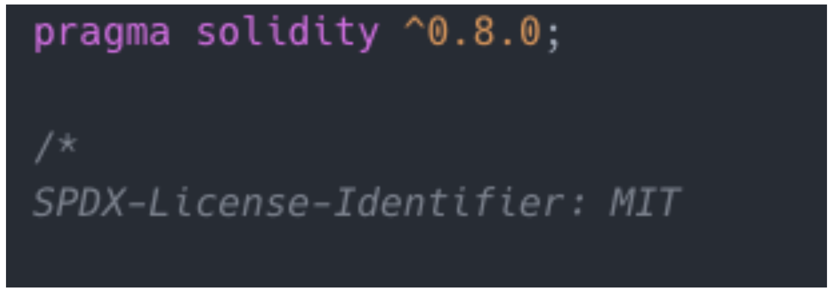
Vulnerability Review

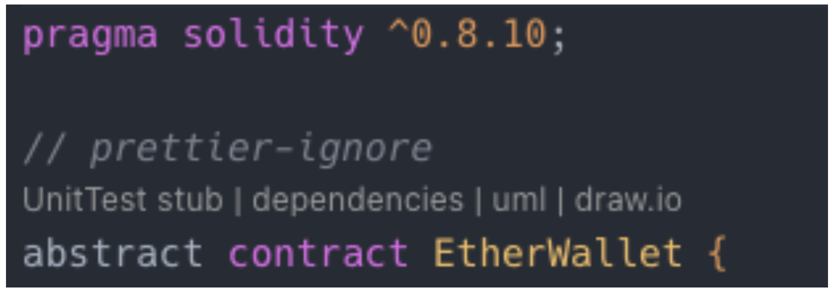
	Number of warnings
Compiler Version	1
Incorrect function state mutability	1
Incorrect state variable status in Whitelisting	1
Queue handling [in loop]	1
State Variable Default Visibility	0
Integer Overflow / Underflow	0
Parity Multisig Bug	0
Callstack Depth Attack	0
Production Node modules security	0
Re-Entrancy	0
Staking user’s money moves expectedly	0

Compiler Version

1

Solidity version used is inconsistent & not fixed. It is highly recommended to update the Solidity version to the latest. Solc frequently releases new compiler versions. 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.

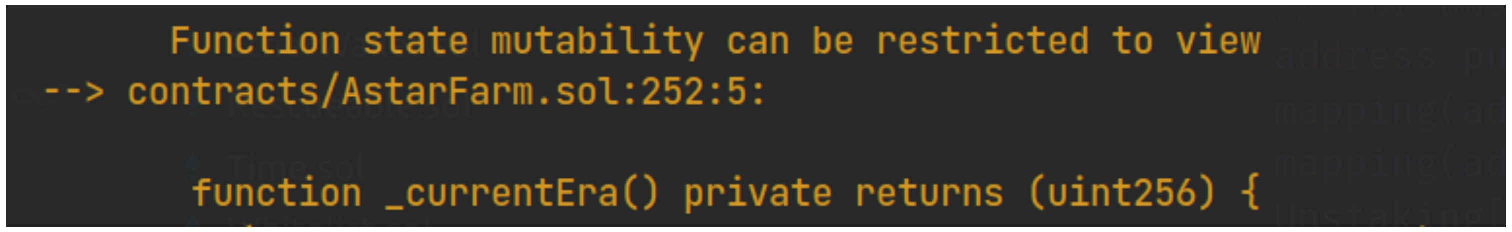

```
pragma solidity ^0.8.0;

/*
SPDX-License-Identifier: MIT
```

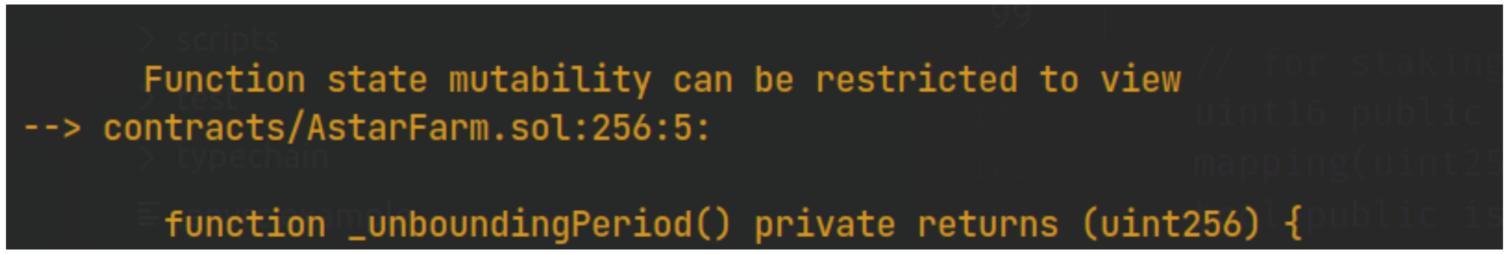

```
pragma solidity ^0.8.10;

// prettier-ignore
UnitTest stub | dependencies | uml | draw.io
abstract contract EtherWallet {
```

Incorrect function state mutability

1

If you make your function pure or view, you can call it for example through web3.js without needing a transaction. Also currently, it will not reduce gas cost when executing on the EVM. However, in the future it may help the Solidity compiler to optimise the contract.

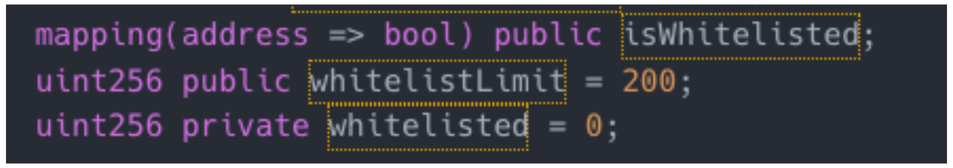

```
    Function state mutability can be restricted to view
 --> contracts/AstarFarm.sol:252:5:

     function _currentEra() private returns (uint256) {
```


```
    Function state mutability can be restricted to view
 --> contracts/AstarFarm.sol:256:5:

     function _unboundingPeriod() private returns (uint256) {
```

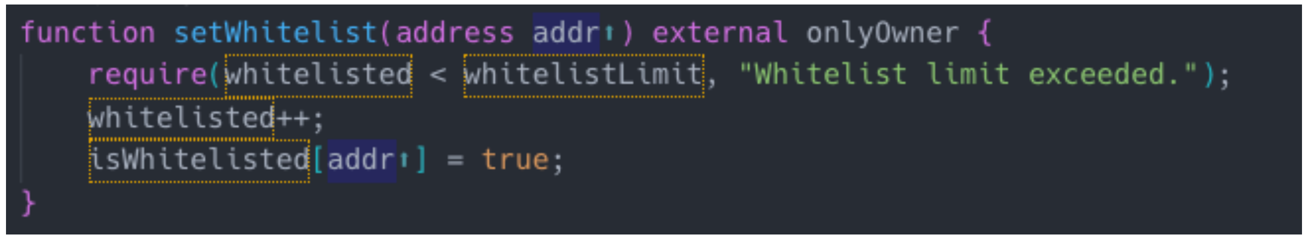
Incorrect state variable status

1

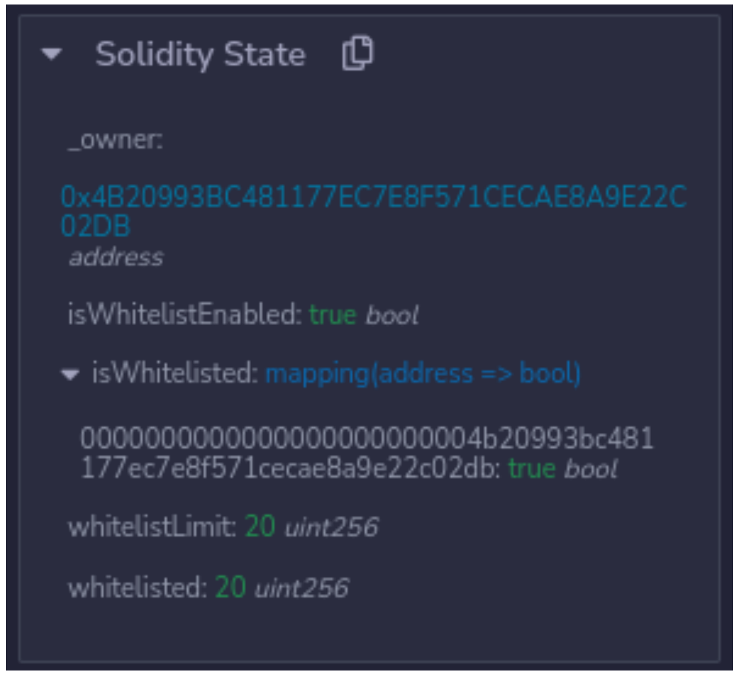
Although this is not in scope (since in the updated code shared later the contract for Whitelist.sol has been changed) , but it is being mentioned as a **Low** level, in case it still exists in the codebase. (since we do not have access to the Git repository).
We have three variables in Whitelist.sol , where count of whitelisted addresses are stored in “whitelisted” variable. When setWhitelist function is called , it first adds the whitelisted variable and then sets the mapping to true. It does not check if the address already exists in the isWhitelisted before increasing the counter. Hence if the same function is called multiple times, it can lead to a state where there is only one address in whitelist but the counter will be full hence cannot add any more address, **leading to blocking address to Whitelist.**



```
mapping(address => bool) public isWhitelisted;
uint256 public whitelistLimit = 200;
uint256 private whitelisted = 0;
```



```
function setWhitelist(address addr) external onlyOwner {
    require(whitelisted < whitelistLimit, "Whitelist limit exceeded.");
    whitelisted++;
    isWhitelisted[addr] = true;
}
```

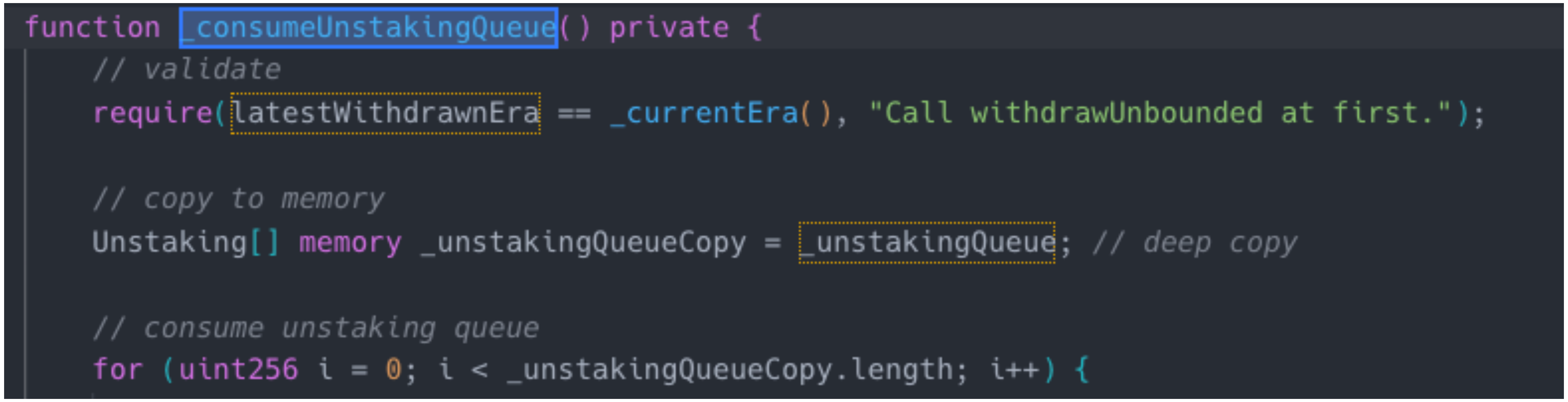


```
Solidity State

_owner:
0x4B20993BC481177EC7E8F571CECAE8A9E22C
02DB
address

isWhitelistEnabled: true bool

isWhitelisted: mapping(address => bool)

  0000000000000000000000004b20993bc481
  177ec7e8f571cecae8a9e22c02db: true bool

whitelistLimit: 20 uint256

whitelisted: 20 uint256
```

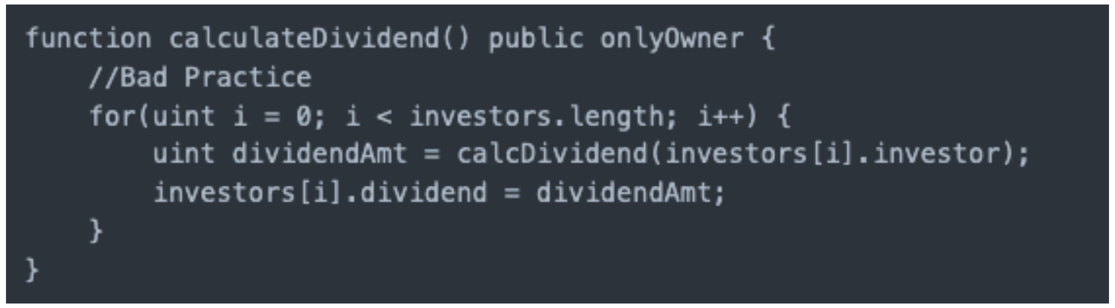
Queue handling [in loop] : Unbounded

1

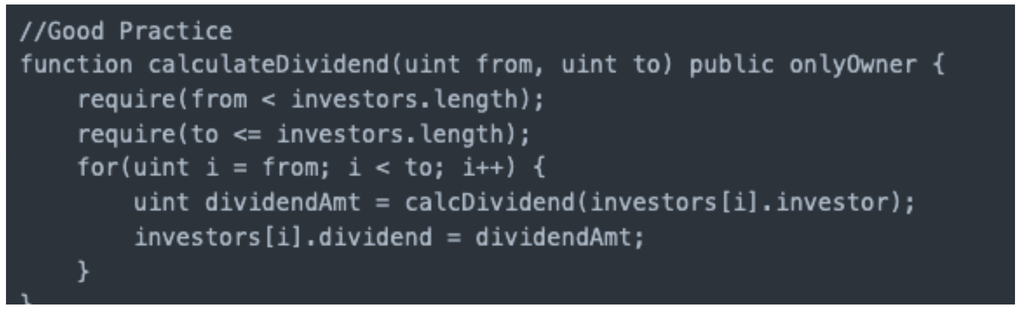
If the loop is updating some state variables of a contract, it should be bounded; otherwise, your contract could get stuck if the loop iteration is hitting the block's gas limit. If a loop is consuming more gas than the block's gas limit, that transaction will not be added to the blockchain; in turn, the transaction would revert. Hence, there is a transaction failure.
You can have unbounded loops for view and pure functions, as these functions are not added into the block; they just read the state from the blockchain when message calls are made to these functions. An example of good and bad practice is given as an example for reference.



```
function _consumeUnstakingQueue() private {
    // validate
    require(latestWithdrawnEra == _currentEra(), "Call withdrawUnbounded at first.");

    // copy to memory
    Unstaking[] memory _unstakingQueueCopy = _unstakingQueue; // deep copy

    // consume unstaking queue
    for (uint256 i = 0; i < _unstakingQueueCopy.length; i++) {
```



```
function calculateDividend() public onlyOwner {
    //Bad Practice
    for(uint i = 0; i < investors.length; i++) {
        uint dividendAmt = calcDividend(investors[i].investor);
        investors[i].dividend = dividendAmt;
    }
}
```



```
//Good Practice
function calculateDividend(uint from, uint to) public onlyOwner {
    require(from < investors.length);
    require(to <= investors.length);
    for(uint i = from; i < to; i++) {
        uint dividendAmt = calcDividend(investors[i].investor);
        investors[i].dividend = dividendAmt;
    }
```