Audit of MAGMA

A report of findings by

Genji Sakamoto

November 23th, 2020

Table of Contents

Executive Summary

This audit report has been written to discover issues and vulnerabilities in the MAGMA smart contracts.

This process included a line by line analysis of the in-scope contracts, optimization analysis, analysis of key functionalities and limiters, and reference against intended functionality.

Audited smart contracts

- MOL.sol
- MAGMA.sol
- MOLStaker.sol
- MAGMAStaker.sol
- Reward.sol

Audit Method

- Static analysis based on source.
- Dynamic analysis by testing deployed ones on Ropsten.

Audit Focus

- Contract logic.
- Vulnerabilities for common and uncommon attacks
- Gas optimization
- Validation for variable limiters
- Transparency for all users.

Conclusion

While auditing the smart contracts for MAGMA project, I found that the idea is very interesting that could attract many users in near future once deployed.

The logic is some tricky, but the whole flow of contracts meet the specification perfectly, except there are some issues I recommend to fix before deployment.

Type of Issues

Title	Description	Issues	SWC ID
Integer Overflow and Underflow	An overflow/underflow happens when an arithmetic operation reaches the maximum or minimum size of a type.	0	SWC-101
Function Incorrectness	Function implementation does not meet the specification, leading to intentional or unintentional vulnerabilities.	0	
Buffer Overflow	An attacker is able to write to arbitrary storage locations of a contract if array of out bound happens.	2	SWC-124
Reentrancy	A malicious contract can call back into the calling contract before the first invocation of the function is finished.	4	SWC-107
Transaction Order Dependence	A race condition vulnerability occurs when code depends on the order of the transactions submitted to it.	0	SWC-114
Timestamp Dependence	Timestamp can be influenced by minors to some degree.	0	SWC-116
Insecure Compiler Version	Using a fixed outdated compiler version or floating pragma can be problematic, if there are publicly disclosed bugs and issues that affect the current compiler version used.	0	SWC-102 SWC-103
Insecure Randomness	Block attributes are insecure to generate random numbers, as they can be influenced by minors to some degree.	0	SWC-120

“tx.origin” for authorization	“tx.origin” should not be used for authorization. Use “msg.sender” instead.	0	SWC-115
Delegate call to Untrusted Calling	Calling into untrusted contracts is very dangerous, the target and arguments provided must be sanitized.	0	SWC-112
State Variable Default Visibility	Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.	2	SWC-108
Function Default Visibility	Functions are public by default. A malicious user is able to make unauthorized or unintended state changes if a developer forgot to set the visibility.	0	SWC-100
Uninitialized Variables	Uninitialized local storage variables can point to other unexpected storage variables in the contract.	0	SWC-109
Assertion Failure	The assert() function is meant to assert invariants. Properly functioning code should never reach a failing assert statement.	0	SWC-110
Deprecated Solidity Features	Several functions and operators in Solidity are deprecated and should not be used as best practice.	0	SWC-111
Unused Variables	Unused variables reduce code quality.	1	

Findings

MOL.sol

State Visibility

Variable visibility is not set for multiple variables.

A public variable is easily accessible to users of the contract, and while no specific issues were found with these variables identified, it is best practice to utilize the most restrictive visibility as possible, unless it is specifically needing to be public.

```
bool inSwapAndLiquify;
bool swapAndLiquifyEnabled;
```

MAGMA.sol

State Visibility

Variable visibility is not set for multiple variables.

A public variable is easily accessible to users of the contract, and while no specific issues were found with these variables identified, it is best practice to utilize the most restrictive visibility as possible, unless it is specifically needing to be public.

```
bool inSwapAndLiquify;
bool swapAndLiquifyEnabled;
bool tradingEnable;
```

Address index for Events

It is better to index addresses of the events for improving the efficiency of getting all events for a user.

```
event RewardStoreAddressUpdated(address rewardStoreAddress);
event StakerAddressUpdated(address stakerAddress);
```

MOLStaker.sol

Address index for Events

It is better to index addresses of the events for improving the efficiency of getting all events for a user.

```
// Events
event Staked(address staker, uint256 amount);
event Unstaked(address staker, uint256 amount);
event Claim(address staker, uint256 amount);
```

Gas Optimization

Now the contract calls claim automatically when user do stake or unstake.

I found that the reason is to assure the correct rewards for stakers, but I recommend to use other flow for gas optimization and correct flow.

We can calculate and save pending rewards every time users stake or unstake. In this way, the user can only stake, unstake or claim as they want , and gas should be optimized as well.

```
if(_stakeMap[_msgSender()].stakedAmount == 0 && _stakeMap[_msgSender()].lastClaimTimestamp == 0) {
    _stakers.push(_msgSender());
    _stakeMap[_msgSender()].lastClaimTimestamp = currentTimestamp;
} else {
    claim();
}
```

```
function unstake(uint256 amount) public {
    require(
        _stakeMap[_msgSender()].stakedAmount >= amount,
        "MOLStaker: unstake amount exceededs the staked amount."
    );

    claim();
```

Vulnerability for Reentrancy

Now the contract reduce the staked amount after call the transfer. It is vulnerability for reentrancy attack.

It should be done before call transfer.

```
require(
    _molContract.transfer(
        _msgSender(),
        amount
    ),
    "MOLStaker: unstake failed."
);

_stakeMap[_msgSender()].stakedAmount = _stakeMap[_msgSender()].stakedAmount.sub(amount);
_totalStakedAmount = _totalStakedAmount.sub(amount);
```

Now the contract changes the last claim timestamp after give reward. It is vulnerability for reentrancy attack.

It should be done before call giveReward.

```
require(
    _rewardContract.giveReward(_msgSender(), rewardAmount),
    "MOLStaker: claim failed."
);

_stakeMap[_msgSender()].lastClaimTimestamp = currentTimestamp;
emit Claim(_msgSender(), rewardAmount);
```

Division by Zero

In the formula for calculating reward, there is a divide operation by _totalStakedAmount. The contract should check if _totalStakedAmount is 0 before calculate reward.

```
uint256 rewardAmount = rewardAmountForStakers.mul(passTime).div(86400).mul(_stakeMap[_msgSender()].stakedAmount).div(_totalStakedAmount);
```

MAGMAStaker.sol

Address index for Events

It is better to index addresses of the events for improving the efficiency of getting all events for a user.

```
// Events
event Staked(address staker, uint256 amount);
event Unstaked(address staker, uint256 amount);
event Claim(address staker, uint256 amount);
```

Gas Optimization

Now the contract calls claim automatically when user do stake or unstake.

I found that the reason is to assure the correct rewards for stakers, but I recommend to use other flow for gas optimization and correct flow.

We can calculate and save pending rewards every time users stake or unstake. In this way, the user can only stake, unstake or claim as they want, and gas should be optimized as well.

```
if(_stakeMap[_msgSender()].stakedAmount == 0 && _stakeMap[_msgSender()].lastClaimTimestamp == 0) {
    _stakers.push(_msgSender());
    _stakeMap[_msgSender()].lastClaimTimestamp = currentTimestamp;
} else {
    claim();
}
```

```
function unstake(uint256 amount) public {
    require(
        _stakeMap[_msgSender()].stakedAmount >= amount,
        "MAGMAStaker: unstake amount exceededs the staked amount."
    );

    claim();
```

Vulnerability for Reentrancy

Now the contract reduce the staked amount after call the transfer. It is vulnerability for reentrancy attack.

It should be done before call transfer.

```
require(
    _lavaContract.transfer(
        _msgSender(),
        amount
    ),
    "MAGMAStaker: unstake failed."
);

_stakeMap[_msgSender()].stakedAmount = _stakeMap[_msgSender()].stakedAmount.sub(amount);
_totalStakedAmount = _totalStakedAmount.sub(amount);
```

Now the contract changes the last claim timestamp after give reward. It is vulnerability for reentrancy attack.

It should be done before call giveReward.

```
require(
    _rewardContract.giveReward(_msgSender(), rewardAmount),
    "MAGMAStaker: claim failed."
);

_stakeMap[_msgSender()].lastClaimTimestamp = currentTimestamp;
emit Claim(_msgSender(), rewardAmount);
```

Division by Zero

In the formula for calculating reward, there is a divide operation by _totalStakedAmount. The contract should check if _totalStakedAmount is 0 before calculate reward.

```
uint256 rewardAmount = rewardAmountForStakers.mul(passTime).div(86400).mul(_stakeMap[_msgSender()].stakedAmount).div(_totalStakedAmount);
```

Reward.sol

Unused Event

The contract customized Ownable contract, and there is no reason to have OwnershipTransferred event.

```
contract RewardOwner is Context {
    address private _molStakerContract;
    address private _magmaStakerContract;
    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
```

Optimization / Best practice

Now the reward contract setup 2 owners (MOLStaker and MAGMAStaker) and allowed only them to give rewards every time user claim.

Now it is checking owner by modifier and also checking staker to call the needed contract's functions. It is not good for gas optimization and clear logic.

```
function getStoreBalance() external onlyOwner view returns (uint256) {
    if(isMolStaker())
        return _magmaContract.balanceOf(address(this));
    else
        return IUniswapV2ERC20(_magmaUniV2Pair).balanceOf(address(this));
}

function giveReward(address recipient, uint256 amount) external onlyOwner returns (bool) {
    if(isMolStaker())
        return _magmaContract.transfer(recipient, amount);
    else
        return IUniswapV2ERC20(_magmaUniV2Pair).transfer(recipient, amount);
}

function withdrawAll(address recipient) external onlyOwner returns (bool) {
    if(isMolStaker()) {
        uint256 balance = _magmaContract.balanceOf(address(this));
        return _magmaContract.transfer(recipient, balance);
    } else {
        uint256 balance = IUniswapV2ERC20(_magmaUniV2Pair).balanceOf(address(this));
        return IUniswapV2ERC20(_magmaUniV2Pair).transfer(recipient, balance);
    }
}
```

I recommend to use separate functions for 2 staker contracts.