FuseFi Smart Contract Initial Audit Report

Overview

VoltFinance by FuseFi

It is a hub of defi tools on the fuse blockchain, providing features of Farming, Lending and Staking.

Scope of Audit

The scope of this audit was to analyze **FuseFi smart contract**'s codebase for quality, security, and correctness.

FuseFi Contracts:

Voltage-Core: https://github.com/voltfinance/voltage-core
Branch: Master
Commit: d881c70e86352d753ec97d29bac6b7d1d8066838

Token-Sale: https://github.com/voltfinance/token-sale-contracts
Branch: Master
Commit: 413fb9a85432fd366f6ca9136580ba2a8872ee37

VoltToken.sol	ERC20 Governance Token
VoltBar.sol	Staking contract where users deposit VOLT and receive xVOLT
MasterChefVoltV2.sol	Staking Vault which rewards users in VOLTs, and contains double rewarder farms for bonus rewards. The VOLT tokens created per second, are distributed to the devs, treasury, investor and pools.
MasterChefVoltV3.sol	Staking Vault on top of V2.
SimpleRewarderPerSec.sol	Used for double rewarder farms,

	rewarding users either in native currency or in bonus token
VoltMakerV2.sol	Converts VOLT LPs into VOLT and sends the VOLT to the VoltBar
TokenSale.sol	Facilitates the token sale where users purchase volts using fuse
VestingVault12.sol	Purchased Volts are vested for the user under two first and second vesting periods, which can be claimed on a per day basis.

Checked Vulnerabilities

We have scanned the smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of **FuseFi smart contracts,** care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the Smart contract is structured in a way that will not result in future problems.

Static Analysis

Static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Mythril, Slither, SmartCheck, Surya, Solhint.

Issue Categories

Every issue in this report has been assigned with a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

TYPE	HIGH	MEDIUM	LOW	INFORMATIONAL
Open	3	9	11	11
Acknowledged	0	0	0	0
Closed	0	0	0	0

Functional Testing Results

Some of the tests performed are mentioned below:

VoltToken

✅should only allow owner to mint token
✅Should be able to transfer tokens
✅Should fail if you try to do bad transfers
✅Should be able to move delegates when transferring tokens
✅User should be able to delegate voting rights to its delegatee

MasterChefVoltV2

✅Should be able to add lp pool
✅Should be able to set/update lp pool data
✅Should be able to deposit
✅Should be able to withdraw with and without rewarder
✅Should be able to emergency withdraw
✅Should work for all pools with intended behavior.
✅Should be able to update emission rate

MasterChefVoltV3

✅Should be able to deposit dummy tokens via init()
✅Should be able to add lp pool
✅Should be able to set/update lp pool data

✅Should be able to deposit
✅Should be able to withdraw with and without rewarder
❌Should be able to emergency withdraw
✅Someone with dummy token can deposit these tokens in MasterChefVoltV2's dummy token pool

SimpleRewarderPerSec

✅Should be able to update pool
✅Should be able to set reward rate
✅Should be able to distribute reward token
✅Should be able to emergency withdraw

VoltBar

✅Should be able to enter VoltBar
✅Should be able to leave VoltBar

VoltMakerv2

✅Should be able to convert token pair to volts
✅Should be able to convert multiple pairs
✅Should be able to convert using bridges

TokenSale

✅Should be able to add purchase limit
✅Should be able to buy tokens by sending FUSE to contract
✅Should be able to buy tokens by calling purchaseTokens
✅Should be able to purchase amount equal to purchase limit
✅Owner can withdraw Fuse from contract
✅Owner can withdraw tokens from contract
✅Should revert if try to purchase after sale has ended
✅Should revert if try to purchase before sale starts
✅Should revert if someone tries to purchase more than purchase limit
❌Purchase tokens will always be successful

VestingVault12

✅Owner should be able to set v12MultiSig
✅v12MultiSig can add token grants

✅Should be able to claim vesting for both schedules

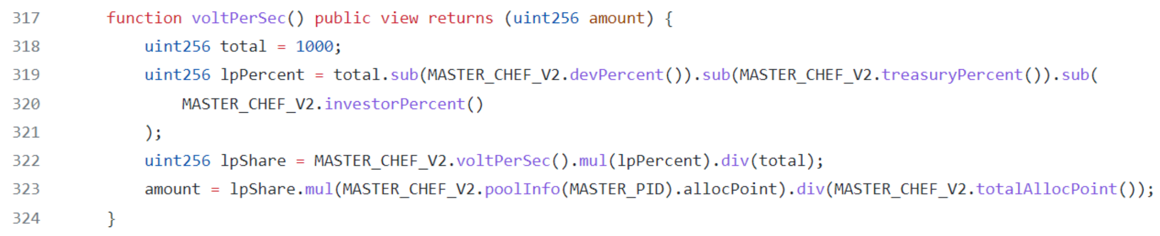
Issues Found

High Severity Issues

MasterChefVoltV3

- **Exploiting voltPerSec by dividing rewards**

MasterChefVoltV3 depends upon MasterChefVoltV2 for its **voltPerSec** reward rate. However, it assumes and builds its logic, by considering that all of the pool rewards of **MASTER_PID** will be harvested/minted to **MasterChefVoltV3**, and doesn't consider the sharing of rewards with multiple users in the same pool.



```
function voltPerSec() public view returns (uint256 amount) {
    uint256 total = 1000;
    uint256 lpPercent = total.sub(MASTER_CHEF_V2.devPercent()).sub(MASTER_CHEF_V2.treasuryPercent()).sub(
        MASTER_CHEF_V2.investorPercent()
    );
    uint256 lpShare = MASTER_CHEF_V2.voltPerSec().mul(lpPercent).div(total);
    amount = lpShare.mul(MASTER_CHEF_V2.poolInfo(MASTER_PID).allocPoint).div(MASTER_CHEF_V2.totalAllocPoint());
}
```

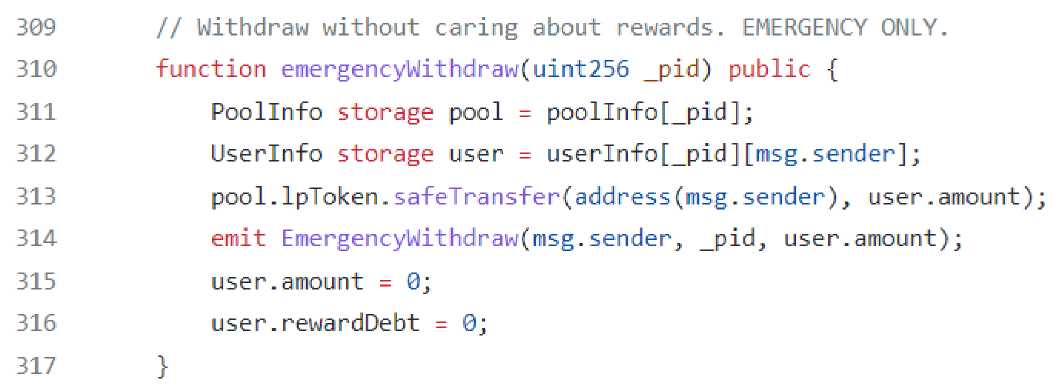
For instance, if another stakes some tokens into the same **MASTER_PID** pool, the total rewards generated will be shared into two users, and may decrease the actual **voltPerSec** reward rate which **MasterChefVoltV3** is referring to for its own reward generation, as a consequence, the **MasterChefVoltV3** may receive/harvest less tokens but still try to provide tokens v3 pools with the total reward rate of **MASTER_PID** pool.

Recommendation: Consider reviewing and verifying the operational and business logic

MasterChefVoltv2

- **emergencyWithdraw allows exploiting Double Rewarder Farms**

Function emergencyWithdraw allows the user to exit from a pool and withdrawing all the staked LP Tokens, without caring about VOLTs rewards. However, the function doesn't call/harvest rewarder's rewards or updates rewarder's state for the particular user. As a result, the user can drain out/ exploit double rewarder farms.



```
    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        pool.lpToken.safeTransfer(address(msg.sender), user.amount);
        emit EmergencyWithdraw(msg.sender, _pid, user.amount);
        user.amount = 0;
        user.rewardDebt = 0;
    }
```

Exploit Scenario:
A user deposits ***x*** number of LP tokens into a **double rewarder farm**. It can use **flash loan** to deposit a large number of tokens. After that it calls **emergencyWithdraw**, taking out all the deposited LP Tokens. Although the user has taken out all the LP Tokens and the vault doesn't have any state for the user, the **rewarder's** state for the user is still intact, and will still serve the same pending rewards whenever the user deposits into the same pool again. Now the user can deposit just a single LP Token and wait for some days, and harvest the

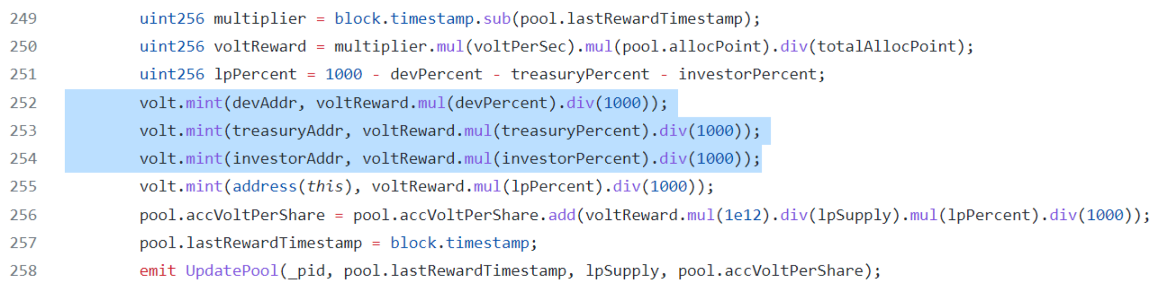
bonus rewards for all the ***x*** number of tokens deposited earlier, instead of 1, hereby possibly draining the rewarder farms.

Recommendation: Consider reviewing and verifying the operational and business logic, and consider harvesting rewards of rewarder in double rewarder farms in the **emergencyWithdraw,** and the rewarder state of user should be updated with 0 amount.

- **Zero address initialization for dev/treasury/investor may lead to unoperational minting/contract**

constructor and **dev/treasury/investor** address **setters**, lacks zero address checks for **_devAddr, _treasuryAddr, _investorAddr**

As the contract tries to mint some VOLTs to **dev/treasury/investor addresses** based on voltPerSec reward rate. If any of these address is accidentally or intentionally set to zero address, the **updatePool** will always revert, as the volt token is an openzeppelin's ERC20 implementation, which doesn't allow any mints over zero addresses.



```
        uint256 multiplier = block.timestamp.sub(pool.lastRewardTimestamp);
        uint256 voltReward = multiplier.mul(voltPerSec).mul(pool.allocPoint).div(totalAllocPoint);
        uint256 lpPercent = 1000 - devPercent - treasuryPercent - investorPercent;
        volt.mint(devAddr, voltReward.mul(devPercent).div(1000));
        volt.mint(treasuryAddr, voltReward.mul(treasuryPercent).div(1000));
        volt.mint(investorAddr, voltReward.mul(investorPercent).div(1000));
        volt.mint(address(this), voltReward.mul(lpPercent).div(1000));
        pool.accVoltPerShare = pool.accVoltPerShare.add(voltReward.mul(1e12).div(lpSupply).mul(lpPercent).div(1000));
        pool.lastRewardTimestamp = block.timestamp;
        emit UpdatePool(_pid, pool.lastRewardTimestamp, lpSupply, pool.accVoltPerShare);
```

Also, since these addresses can only be changed by existing addresses. It will not be possible to change them later as nobody controls zero address, hence making the minting and in turn the entire contract unoperational.

Recommendation: Consider reviewing and verifying the operational and business logic and adding zero address checks for dev, investor and treasury addresses.

Medium Severity Issues

- **Old Compiler version**

  VestIngVault12 contract **(**using 0.4.24**)** and **Voltage-core** Contracts **(**using 0.6.12**)** are using old version of solidity, solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks.

  Recommendation: Use the latest compiler version in order to avoid bugs introduced in older versions.

VestingVault12

- **Lack of test coverage:**

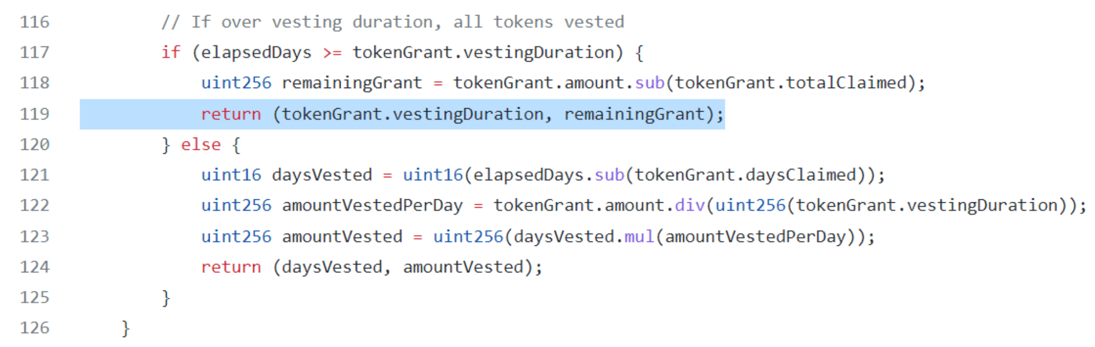
  [#L148] function removeTokenGrant is used to remove a grant and transfer the funds which were not vested back to the **v12MultiSig** and the function can only be called by **v12MultiSig**. The intended operational logic is that the **v12MultiSig** is supposed to be changed to **TokenSale** contract after the initialization of the vault, in order to allow **TokenSale** contract to add token grants in the vault. However, **TokenSale** doesn't contain any function to call vault's **removeTokenGrant,** as a consequence a grant will never be removed from the vault.

  For the same reason, vault's **changeMultiSig** will be unoperational, as the existing **TokenSale** contract doesn't contain any function to change multisig to a new address.

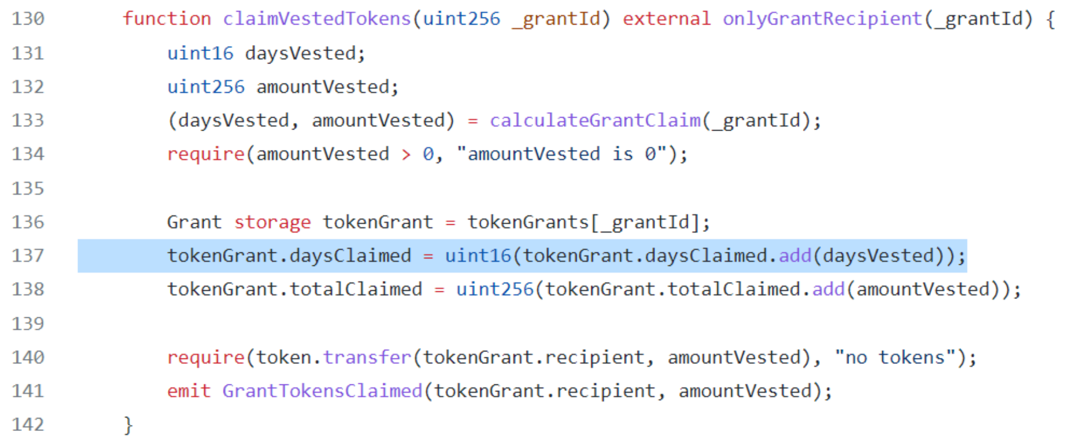
  Recommendation: Consider reviewing and verifying the operational and business logic

- **Incorrect Vesting Days Calculation**

  [#L100] function calculateGrantClaim, calculates and returns the vested days and amount for a particular grant. However, it returns incorrect vested days for some scenarios, which may lead to incorrect **daysClaimed** value for the grant, while claiming vested tokens.



```
        // If over vesting duration, all tokens vested
        if (elapsedDays >= tokenGrant.vestingDuration) {
            uint256 remainingGrant = tokenGrant.amount.sub(tokenGrant.totalClaimed);
            return (tokenGrant.vestingDuration, remainingGrant);
        } else {
            uint16 daysVested = uint16(elapsedDays.sub(tokenGrant.daysClaimed));
            uint256 amountVestedPerDay = tokenGrant.amount.div(uint256(tokenGrant.vestingDuration));
            uint256 amountVested = uint256(daysVested.mul(amountVestedPerDay));
            return (daysVested, amountVested);
        }
    }
```



```
    function claimVestedTokens(uint256 _grantId) external onlyGrantRecipient(_grantId) {
        uint16 daysVested;
        uint256 amountVested;
        (daysVested, amountVested) = calculateGrantClaim(_grantId);
        require(amountVested > 0, "amountVested is 0");

        Grant storage tokenGrant = tokenGrants[_grantId];
        tokenGrant.daysClaimed = uint16(tokenGrant.daysClaimed.add(daysVested));
        tokenGrant.totalClaimed = uint256(tokenGrant.totalClaimed.add(amountVested));

        require(token.transfer(tokenGrant.recipient, amountVested), "no tokens");
        emit GrantTokensClaimed(tokenGrant.recipient, amountVested);
    }
```

For instance,
Let’s assume, an active grant of 100 tokens has a vesting duration of 10 days. So, per day, the recipient is eligible to claim 10 vested tokens, considering there is no cliff(0 days).

Let’s say, 5 days have passed since the grant was started. Now the recipient wants to claim vested tokens, **calculateGrantClaim** will return

daysVested:	**5**
amountVested:	**50**

grants.daysClaimed will be 5

grants.totalClaimed will be 50

Now, let’s say the user came back after 10 days of claiming the last grant, meaning the grant duration is over. Now the recipient wants to claim remaining vested tokens, **calculateGrantClaim** will return

daysVested: **10** (the entire grant duration, and not the unclaimed days)

amountVested: **100 - 50 = 50** (total grant amount - total claimed amount)

grants.daysClaimed will be 5 + 10 = 15

grants.totalClaimed will be 50 + 50 = 100

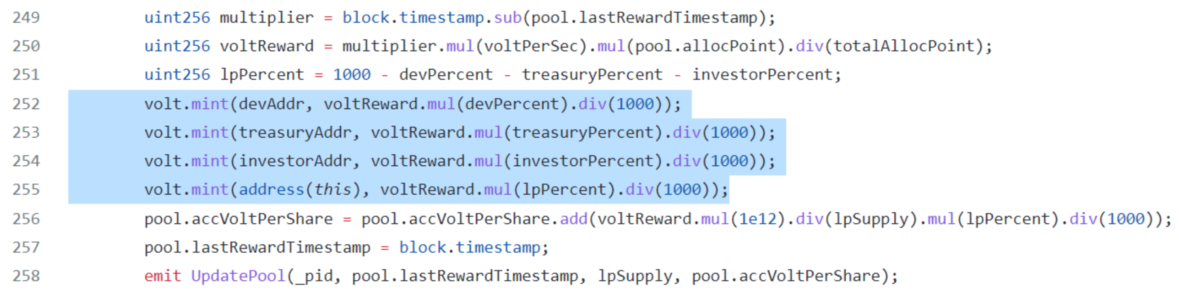
So, even though the grant duration was 10 days, the grant’s **daysClaimed** is now pointing to 15 days.

Recommendation: Consider reviewing and verifying the operational and business logic. Possible solution might be to return **tokenGrant.vestingDuration - tokenGrant.daysClaimed** as the vested days if the vesting duration is over

MasterChefVoltv2

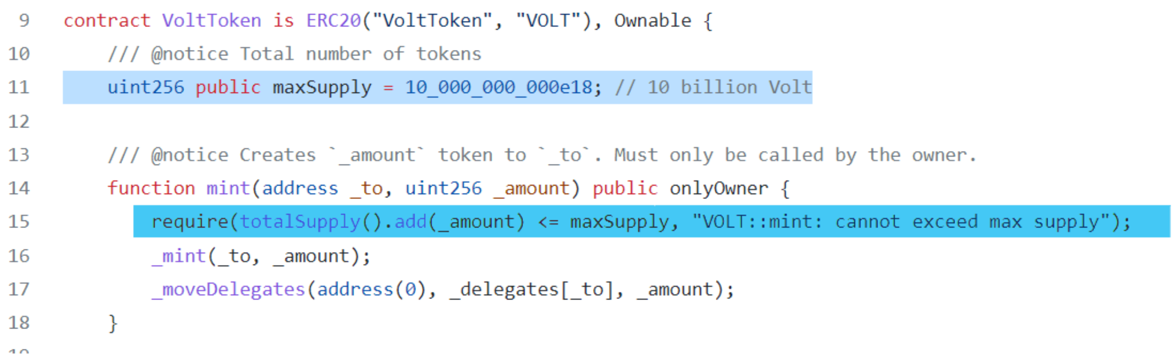
- **Limited Supply can lead to unoperational minting/contract**

V2 generates/mints VOLTs based on the defined **voltPerSec** reward rate.



```
        uint256 multiplier = block.timestamp.sub(pool.lastRewardTimestamp);
        uint256 voltReward = multiplier.mul(voltPerSec).mul(pool.allocPoint).div(totalAllocPoint);
        uint256 lpPercent = 1000 - devPercent - treasuryPercent - investorPercent;
        volt.mint(devAddr, voltReward.mul(devPercent).div(1000));
        volt.mint(treasuryAddr, voltReward.mul(treasuryPercent).div(1000));
        volt.mint(investorAddr, voltReward.mul(investorPercent).div(1000));
        volt.mint(address(this), voltReward.mul(lpPercent).div(1000));
        pool.accVoltPerShare = pool.accVoltPerShare.add(voltReward.mul(1e12).div(lpSupply).mul(lpPercent).div(1000));
        pool.lastRewardTimestamp = block.timestamp;
        emit UpdatePool(_pid, pool.lastRewardTimestamp, lpSupply, pool.accVoltPerShare);
```

However, the supply to mint is capped, and can’t exceed more than maxSupply.



```
contract VoltToken is ERC20("VoltToken", "VOLT"), Ownable {
    /// @notice Total number of tokens
    uint256 public maxSupply = 10_000_000_000e18; // 10 billion Volt

    /// @notice Creates `_amount` token to `_to`. Must only be called by the owner.
    function mint(address _to, uint256 _amount) public onlyOwner {
        require(totalSupply().add(_amount) <= maxSupply, "VOLT::mint: cannot exceed max supply");
        _mint(_to, _amount);
        _moveDelegates(address(0), _delegates[_to], _amount);
    }
```

Thus any new reward generation beyond maxSupply will fail, which will in turn make v2 non-functional or operational, as none of the critical operations will work.

Recommendation: Consider reviewing and verifying the operational and business logic

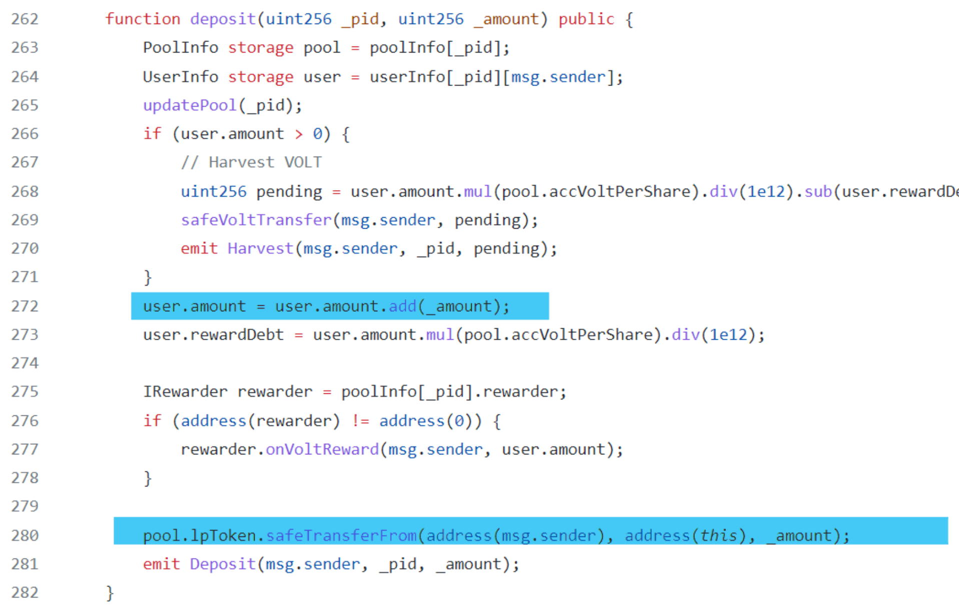
- **Possible Reentrancy Issues**

The critical operations: **deposit** and **Withdraw**, can lead to reentrancy scenarios, in double rewarder farms, considering the fact that recipient/caller can make a reentrant call if the rewarder rewards in native currency. The reentrant calls may produce incorrect results in edge case scenarios.

Recommendation: Consider using openzeppelin's ReentrancyGuard for **deposit** and **withdraw** in order to avoid risks that can generate from edge case/ cross function reentrancy scenarios.

- **Missing best practice to handle deflationary tokens**

Function deposit, allows a user to deposit lpTokens into a pool. However, it doesn't follow the best practice for handling deflationary tokens. If an LP Token is an ERC20 deflationary token(fee deducing tokens), the contract will receive less funds from the user, but update the critical state variables **amount** and **rewardDebt**, based on initial values, rather than considering the funds which were actually received. It will allow the user to extract more rewards than intended.

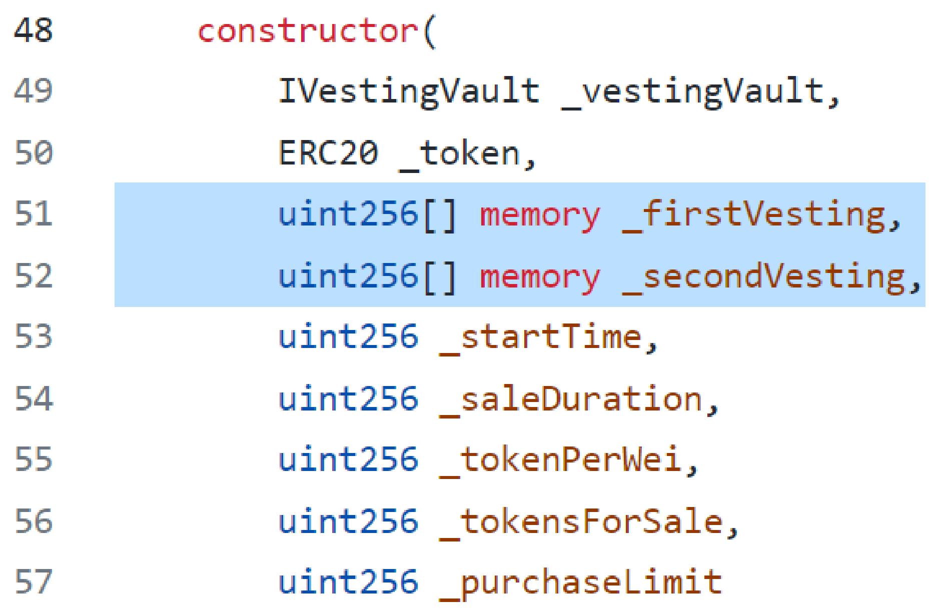


```
function deposit(uint256 _pid, uint256 _amount) public {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    updatePool(_pid);
    if (user.amount > 0) {
        // Harvest VOLT
        uint256 pending = user.amount.mul(pool.accVoltPerShare).div(1e12).sub(user.rewardDebt);
        safeVoltTransfer(msg.sender, pending);
        emit Harvest(msg.sender, _pid, pending);
    }
    user.amount = user.amount.add(_amount);
    user.rewardDebt = user.amount.mul(pool.accVoltPerShare).div(1e12);

    IRewarder rewarder = poolInfo[_pid].rewarder;
    if (address(rewarder) != address(0)) {
        rewarder.onVoltReward(msg.sender, user.amount);
    }

    pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
    emit Deposit(msg.sender, _pid, _amount);
}
```

Recommendation: Consider following the best practice as:
Check contract funds prior and after transferring tokens from the user, and subtracting the newer balance from the older balance. It will provide the actual amount of LPTokens that were received during the deposit. Now this amount should be considered to update the critical state variables **amount** and **rewardDebt** for the user.

TokenSale

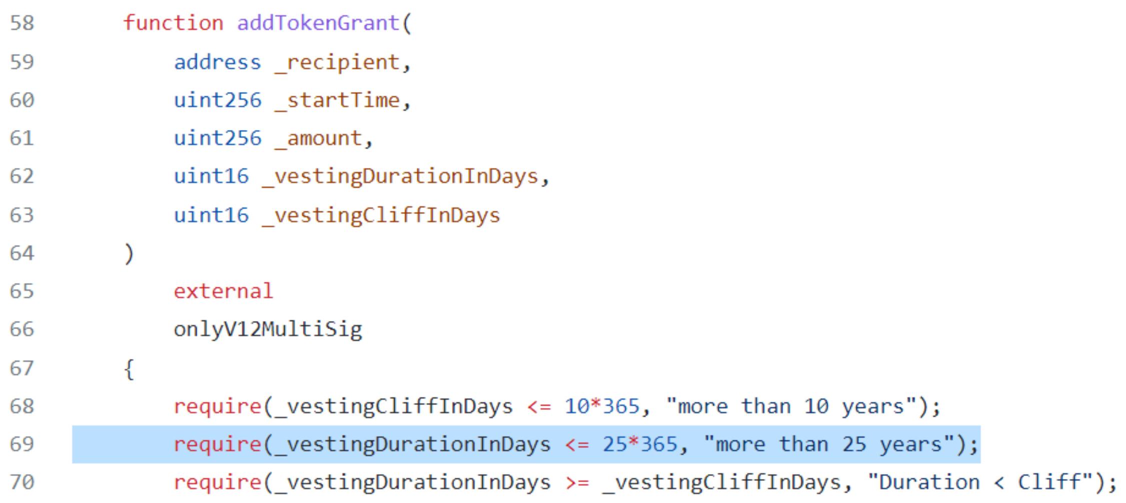
- **Missing Important Checks**
  [#L48] constructor: _firstVesting[0] and **_secondVesting[0]** defines the duration in days for the first and second vesting respectively. However, constructor lacks important checks for them, which may lead to incorrect initialization and produce incorrect results.



```
    constructor(
        IVestingVault _vestingVault,
        ERC20 _token,
        uint256[] memory _firstVesting,
        uint256[] memory _secondVesting,
        uint256 _startTime,
        uint256 _saleDuration,
        uint256 _tokenPerWei,
        uint256 _tokensForSale,
        uint256 _purchaseLimit
```

Some important checks to consider:

1. **Zero Duration: _firstVesting[0]** and **_secondVesting[0]** can be initialized with 0 vesting duration, as a consequence the user will end up paying **fuse** amount but no VOLTs will be vested for it.
2. **Upper Bound Checks:** As it is hardcoded in the function **addTokenGrant** in Vesting vault, that duration can't exceed 25 years. However, there is no such check added in the token sale constructor, hence it is possible to initialize token sale with vesting periods of duration more than 25 years. If initialized with more than 25 years, the token purchase transactions will always revert.



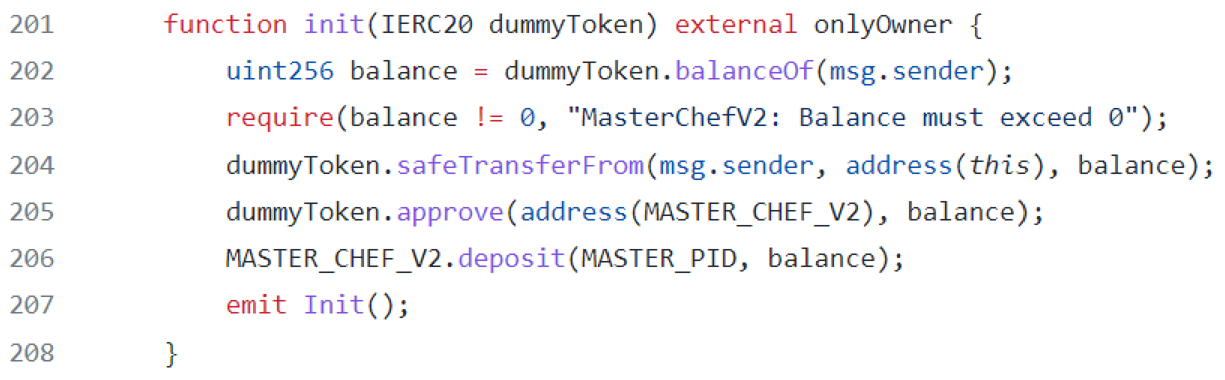
```
    function addTokenGrant(
        address _recipient,
        uint256 _startTime,
        uint256 _amount,
        uint16 _vestingDurationInDays,
        uint16 _vestingCliffInDays
    )
        external
        onlyV12MultiSig
    {
        require(_vestingCliffInDays <= 10*365, "more than 10 years");
        require(_vestingDurationInDays <= 25*365, "more than 25 years");
        require(_vestingDurationInDays >= _vestingCliffInDays, "Duration < Cliff");
```

Recommendation: Consider adding the required checks

MasterChefVoltV3

- **Missing Input Validation**

  Function init, allows the owner to deposit dummyToken to the MASTER_PID pool of MasterChefVoltV2. However, there is no check to make sure the **dummyToken** should be the same as the lpToken of **MASTER_PID** pool, as a consequence, if supplied an incorrect token, all the owner's tokens will be forever stuck in the contract, as there is no function to recover ERC20 tokens.



```
    function init(IERC20 dummyToken) external onlyOwner {
        uint256 balance = dummyToken.balanceOf(msg.sender);
        require(balance != 0, "MasterChefV2: Balance must exceed 0");
        dummyToken.safeTransferFrom(msg.sender, address(this), balance);
        dummyToken.approve(address(MASTER_CHEF_V2), balance);
        MASTER_CHEF_V2.deposit(MASTER_PID, balance);
        emit Init();
    }
```

Recommendation: Consider adding a check so as to make sure the input address of **dummyToken** should be the same as the **MASTER_PID** pool's lpToken.

- **Not updating existing pools prior to adding/modifying a pool**

  [#L220] function add & **[#L254] function set**, don't all the existing update pools prior to adding a new pool or modifying an existing pool, which may affect the reward rate for the existing pools, as adding and modifying a pool changes the **totalAllocPoint**, which is a crucial factor that determines the reward rate of a particular pool.

  For instance,
  Let's assume voltPerSec is 1 token, thus 100 seconds will generate a reward of 100 tokens. Now, let's assume there are 4 pools with allocation point of 25 each. So, the 25 tokens will be distributed to each pool.

  However, the pools were not updated before adding a new pool, and a new pool again with the same allocation point of 25 gets added. Now 100 rewards are supposed to be distributed to 5 pools, with 20 tokens to each pool.

  This can be prevented if we update the existing pools, to distribute the generated rewards with the current reward rate, and then add/modify a pool.

  Recommendation: Consider reviewing and verifying the operational and business logic.

Low Severity Issues

- **Missing Zero Address Checks**

  Contracts lack zero address checks, hence are prone to be initialized with zero addresses.

  MasterChefVoltV2 contract lacks zero address check for **_volt**

MasterChefVoltV3 contract lacks zero address check for **_MASTER_CHEF_V2, _volt**

VoltBar contract lacks zero address check for **_volt**

VoltMakerV2 contract lacks zero address check for **_factory, _bar, _volt, _wfuse**

Recommendation: Consider adding zero address checks in order to avoid risks of incorrect contract initializations.

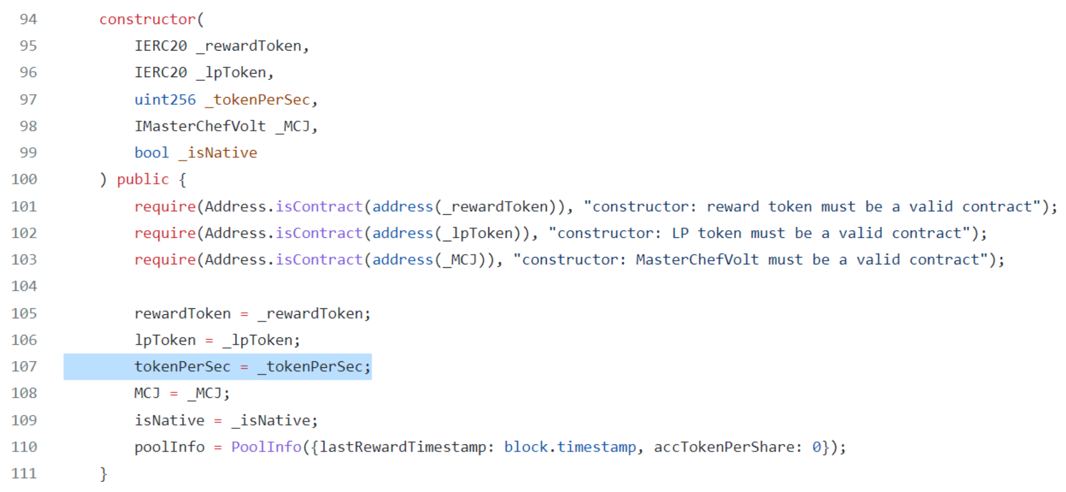
SimpleRewarderPerSec

- **Missing SafeMath operations for arithmetic calculations**

  Contract uses SafeMath library for safe arithmetic calculations as the 0.6.12 compiler version doesn't have any built-in safe arithmetic checks. However, some instances of unsafe arithmetic calculations have been reported.

  Recommendation: Consider using SafeMath for all the arithmetic calculations to increase code readability and reduce risks that may arise from any edge case scenario.

- **Missing Important Checks**
  [#L94] constructor lacks value checks for **_tokenPerSec** which may result in an incorrect and undesired initialization.



```
constructor(
    IERC20 _rewardToken,
    IERC20 _lpToken,
    uint256 _tokenPerSec,
    IMasterChefVolt _MCJ,
    bool _isNative
) public {
    require(Address.isContract(address(_rewardToken)), "constructor: reward token must be a valid contract");
    require(Address.isContract(address(_lpToken)), "constructor: LP token must be a valid contract");
    require(Address.isContract(address(_MCJ)), "constructor: MasterChefVolt must be a valid contract");

    rewardToken = _rewardToken;
    lpToken = _lpToken;
    tokenPerSec = _tokenPerSec;
    MCJ = _MCJ;
    isNative = _isNative;
    poolInfo = PoolInfo({lastRewardTimestamp: block.timestamp, accTokenPerShare: 0});
}
```

Recommendation: Consider adding the required checks defining a lower and upper bound for the value.

MasterChefVoltV2

- **Critical Address change**

When privileged roles are being changed, it is recommended to follow a two-step approach: 1) The current privileged role proposes a new address for the change 2) The newly proposed address then claims the privileged role in a separate transaction. This two-step change allows accidental proposals to be corrected instead of leaving the system operationally with no/malicious privileged role. (Ref: Security Pitfalls: 162)

However, functions dev, setTreasuryAddr, setInvestorAddr functions set critical addresses following a one step process, which may lead to accidental loss of access control over privileged roles

Recommendation: Consider switching to a two-step process for transferring critical and privileged roles

- **Missing Important Checks**

  constructor lacks value checks for **_startTimestamp**, which is a crucial parameter that defines when the reward generation starts for a particular pool. The contract can however be initialized with **_startTimestamp** with a time in the past.

  constructor and **updateEmissionRate** lacks value checks for **_voltPerSec**, which is a crucial parameter that defines how the rewards will be generated. The contract can however be initialized with unintended and incorrect **_voltPerSec.**

  Recommendation: Consider reviewing and verifying the operational and business logic, and consider adding necessary checks so as to avoid risks of incorrect contract initialization.

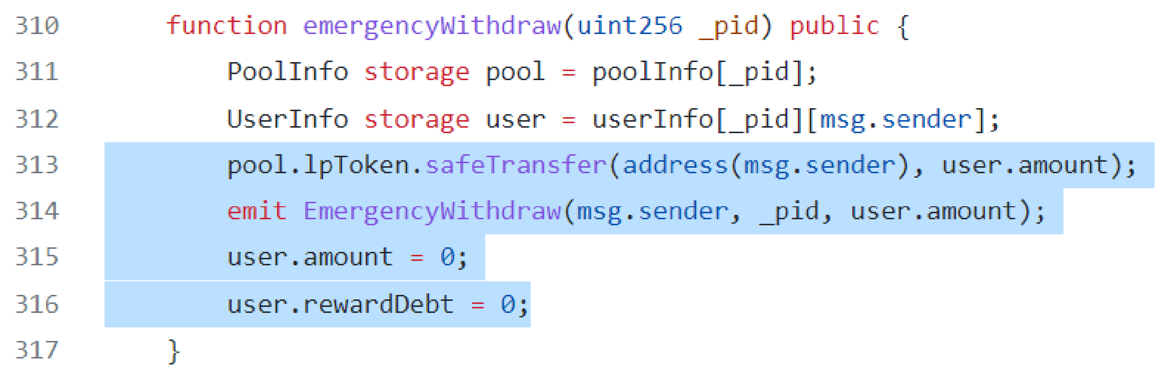
- **Missing Events for Critical Operations**

  The contract does not emit events for critical operations **dev/treasury/investor percent** and **address** setters. Missing events will make it difficult for the off-chain monitoring tools to track critical parameters and operations off-chain.

  Recommendation: Consider adding events for critical operations.

- **External call before updating critical state variables.**

  Function emergencyWithdraw makes an external call prior to updating the critical state variables amount and rewardDebt for the caller.



```
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        pool.lpToken.safeTransfer(address(msg.sender), user.amount);
        emit EmergencyWithdraw(msg.sender, _pid, user.amount);
        user.amount = 0;
        user.rewardDebt = 0;
    }
```

In case of a malicious lpToken, the external call may lead to reentrancy and caller may exploit the contract funds.

Recommendation: Consider changing the critical state prior to making an external call thus following the Check-Effects-Interaction pattern. Openzeppelin's ReentrancyGuard may be used as well.

VestingVault12 & TokenSale

- **Use of token contracts instead of interfaces**

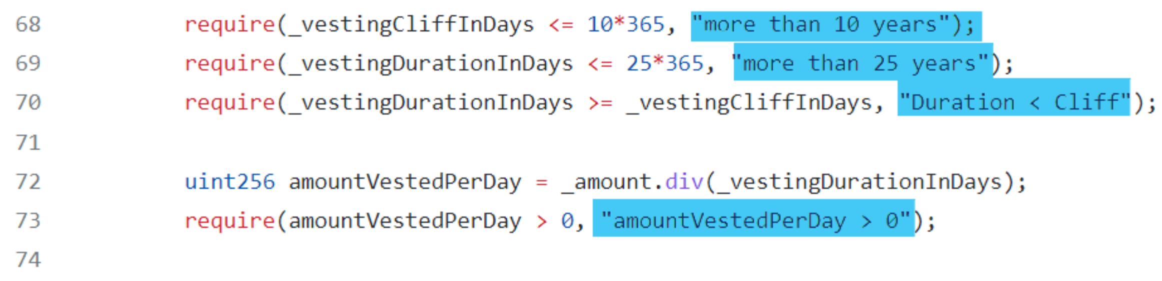
  Contract deals with ERC20 token, instead of using token interface **VestingVault12** & **TokenSale** contract imports the complete token contract which comes with already imported libraries and contracts.

  Recommendation: Token interfaces can be used instead of token complete contract.

VestingVault12

- **Meaningless Error Statement**

  The error statements used in the contract don't convey a meaningful information or message.



```
        require(_vestingCliffInDays <= 10*365, "more than 10 years");
        require(_vestingDurationInDays <= 25*365, "more than 25 years");
        require(_vestingDurationInDays >= _vestingCliffInDays, "Duration < Cliff");

        uint256 amountVestedPerDay = _amount.div(_vestingDurationInDays);
        require(amountVestedPerDay > 0, "amountVestedPerDay > 0");

```

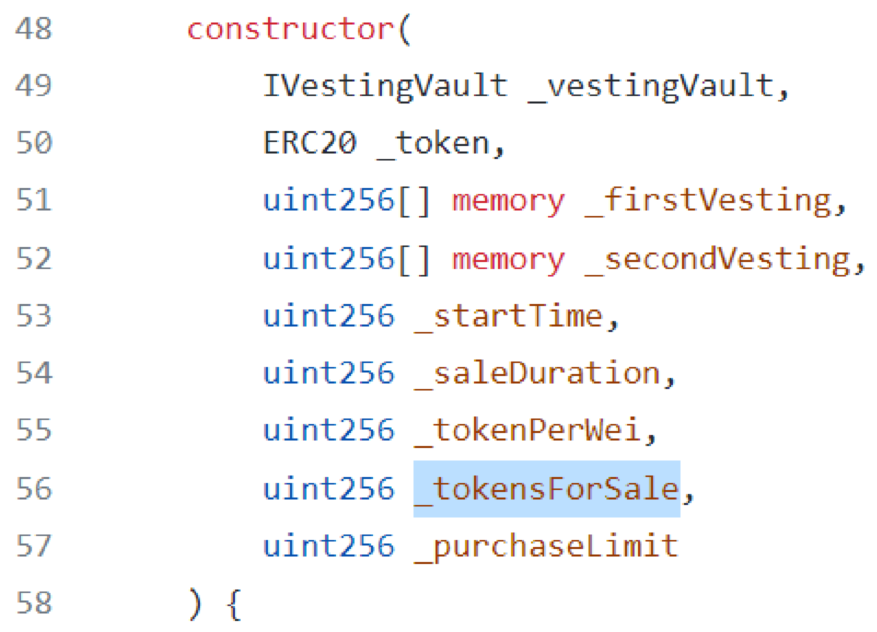
Recommendation: Consider adding meaningful error statements in order to convey easy to understand messages and improve code readability at the same time.

- **Missing Input Validation**
  Functions don't perform input validation for **_grantId** input parameter, as a consequence, the function may produce incorrect or inconsistent results.

  Recommendation: Consider adding checks/modifier in all the functions dealing with **_grantId**, making sure that the input grant id should be less than **totalVestingCount**.

TokenSale

- **Missing Important Checks**
  [#L48] constructor initializes the token sale with **_tokensForSale**, which defines the number of available tokens for sale. However, there is no such check to make sure the contract will have the required tokens to vest and no token purchase will fail.



```
    constructor(
        IVestingVault _vestingVault,
        ERC20 _token,
        uint256[] memory _firstVesting,
        uint256[] memory _secondVesting,
        uint256 _startTime,
        uint256 _saleDuration,
        uint256 _tokenPerWei,
        uint256 _tokensForSale,
        uint256 _purchaseLimit
    ) {
```

Recommendation: Consider transferring tokens equal to **_tokensForSale** from the deployer to the contract at the time of contract initialization itself.

Informational

- **Public** functions never used by the contract internally should be declared **external** to save gas

- **Floating Pragma:**

  Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly.
  Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

  Recommendation: Lock the pragma version for the compiler version that is chosen.

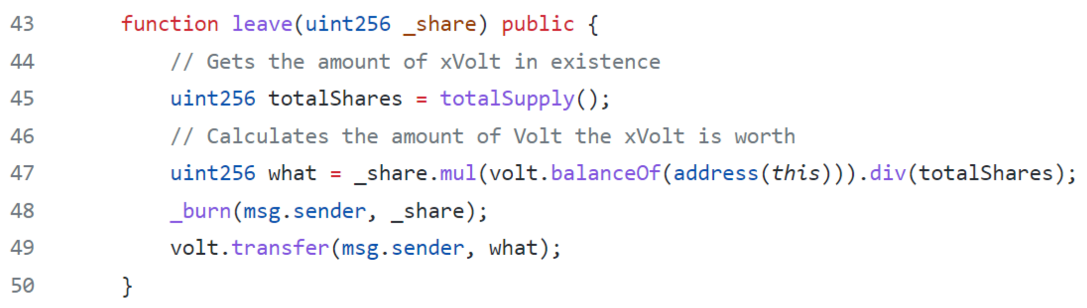
- Trader Joe's references have been reported multiple times in multiple contracts.

Recommendation: Consider replacing trader joe's references with volt and voltage finance to increase code readability.

- As most of the logic has been built around 18 decimal tokens. It is best advised to make sure the contracts deal with 18 decimal tokens. Tokens with different decimals may produce incorrect results in edge case scenarios.

VoltBar

- **[#L43] function leave:** Function contains no checks for zero total supply which may result in **SafeMath: division by zero** reverts

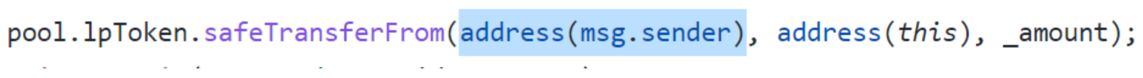


```
43      function leave(uint256 _share) public {
44          // Gets the amount of xVolt in existence
45          uint256 totalShares = totalSupply();
46          // Calculates the amount of Volt the xVolt is worth
47          uint256 what = _share.mul(volt.balanceOf(address(this))).div(totalShares);
48          _burn(msg.sender, _share);
49          volt.transfer(msg.sender, what);
50      }
```

Recommendation: Consider adding the required checks to handle the mentioned case with custom error statements.

MasterChefVoltv2

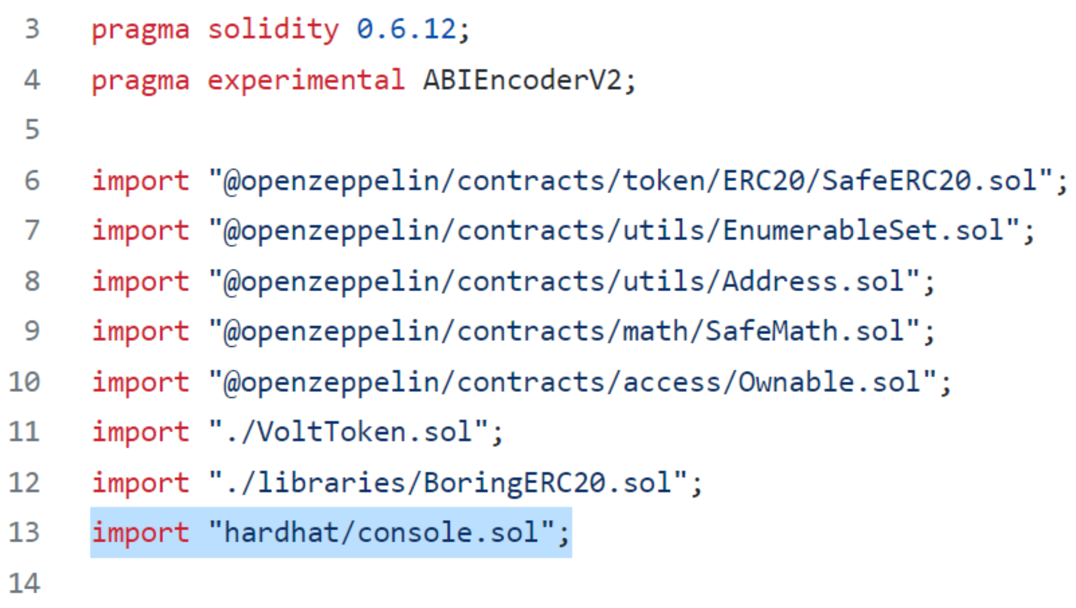
- **[#L280, #L305, #L313] msg.sender** is wrapped in address type, which is not required.



```
pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
```

Recommendation: msg.sender global variable returns address type value that can be used without wrapping it with address().

- Contract imports hardhat debugging libraries, which is supposed to be removed prior to deployment of contract

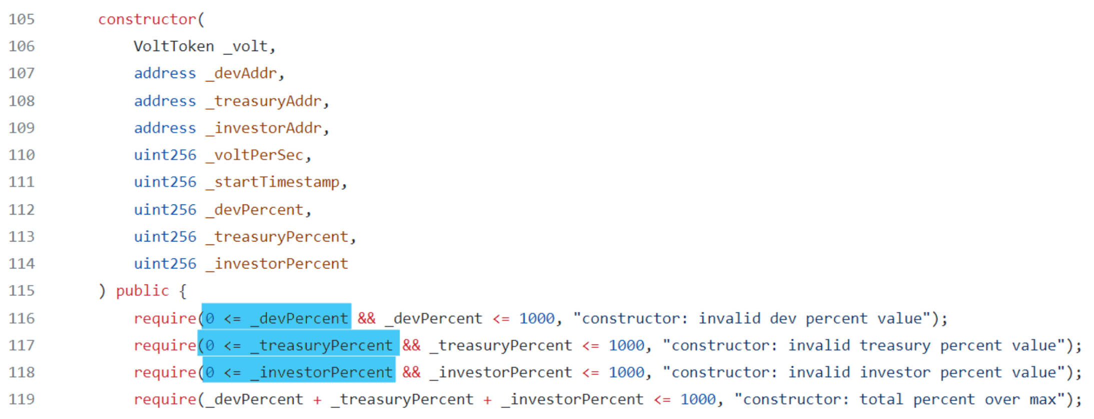


```
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
import "@openzeppelin/contracts/utils/EnumerableSet.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "./VoltToken.sol";
import "./libraries/BoringERC20.sol";
import "hardhat/console.sol";

```

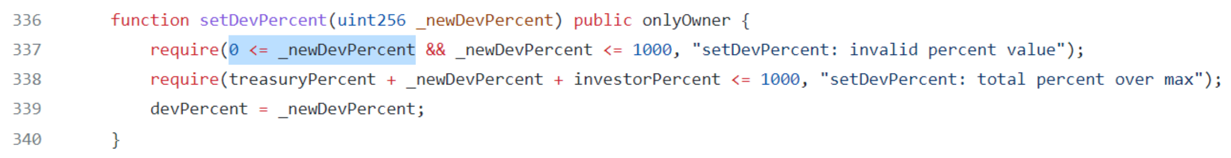
Recommendation: Consider removing hardhat debugging libraries.

- **Unnecessary value checks**

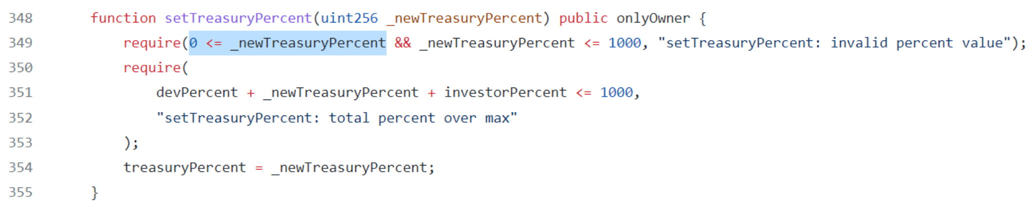
constructor & **dev/treasury/investor** percent setter functions perform input validation, by checking that input percent should be **>=0**. However, these are not needed as **uint256 values** will always be **>=0**



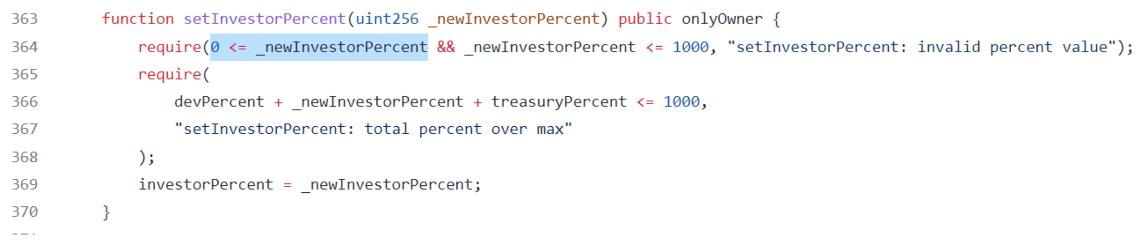
```
constructor(
    VoltToken _volt,
    address _devAddr,
    address _treasuryAddr,
    address _investorAddr,
    uint256 _voltPerSec,
    uint256 _startTimestamp,
    uint256 _devPercent,
    uint256 _treasuryPercent,
    uint256 _investorPercent
) public {
    require(0 <= _devPercent && _devPercent <= 1000, "constructor: invalid dev percent value");
    require(0 <= _treasuryPercent && _treasuryPercent <= 1000, "constructor: invalid treasury percent value");
    require(0 <= _investorPercent && _investorPercent <= 1000, "constructor: invalid investor percent value");
    require(_devPercent + _treasuryPercent + _investorPercent <= 1000, "constructor: total percent over max");
```



```
function setDevPercent(uint256 _newDevPercent) public onlyOwner {
    require(0 <= _newDevPercent && _newDevPercent <= 1000, "setDevPercent: invalid percent value");
    require(treasuryPercent + _newDevPercent + investorPercent <= 1000, "setDevPercent: total percent over max");
    devPercent = _newDevPercent;
}
```



```
function setTreasuryPercent(uint256 _newTreasuryPercent) public onlyOwner {
    require(0 <= _newTreasuryPercent && _newTreasuryPercent <= 1000, "setTreasuryPercent: invalid percent value");
    require(
        devPercent + _newTreasuryPercent + investorPercent <= 1000,
        "setTreasuryPercent: total percent over max"
    );
    treasuryPercent = _newTreasuryPercent;
}
```

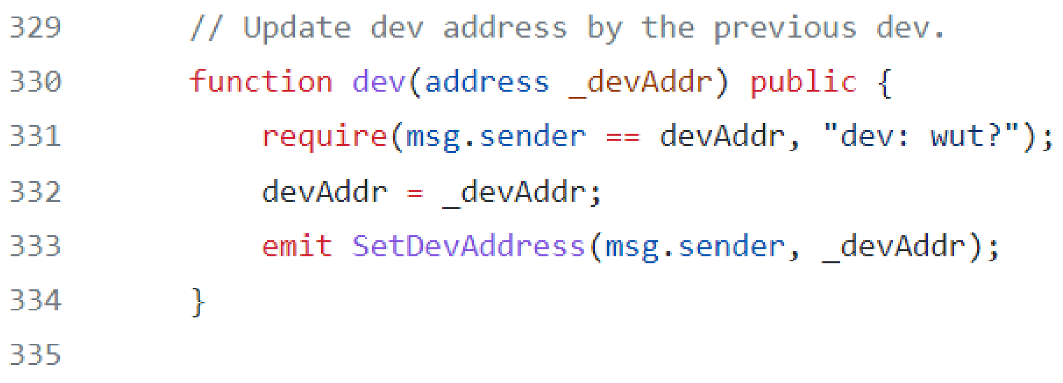


```
function setInvestorPercent(uint256 _newInvestorPercent) public onlyOwner {
    require(0 <= _newInvestorPercent && _newInvestorPercent <= 1000, "setInvestorPercent: invalid percent value");
    require(
        devPercent + _newInvestorPercent + treasuryPercent <= 1000,
        "setInvestorPercent: total percent over max"
    );
    investorPercent = _newInvestorPercent;
}
```

Recommendation: Consider removing unnecessary checks

- **Meaningless function names**

  Function dev allows the existing devAddr to assign a new devAddr. However the function name doesn't convey meaningful information.



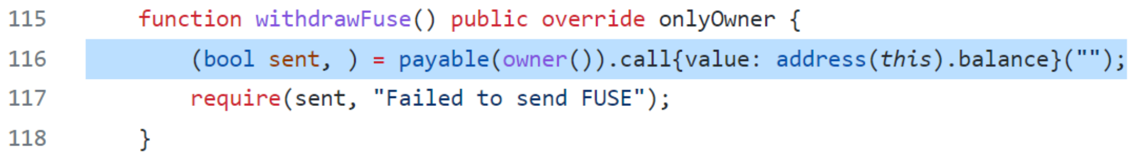
```
    // Update dev address by the previous dev.
    function dev(address _devAddr) public {
        require(msg.sender == devAddr, "dev: wut?");
        devAddr = _devAddr;
        emit SetDevAddress(msg.sender, _devAddr);
    }

```

Recommendation: Consider giving meaningful names to critical functions

TokenSale

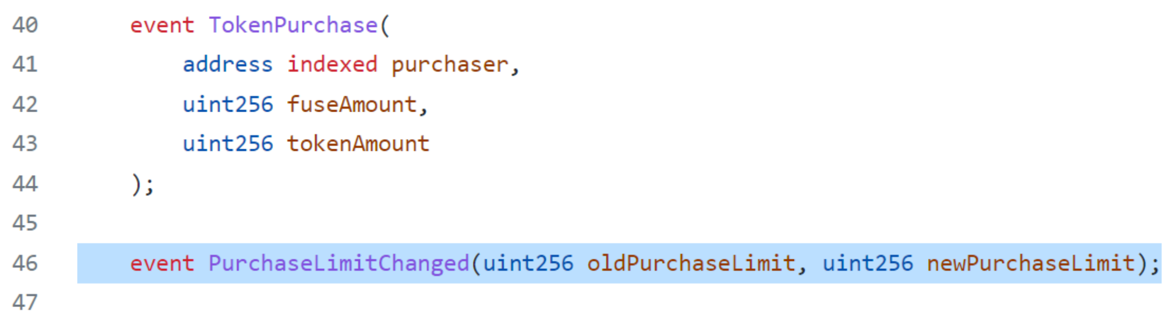
- **Unnecessary Payable Keyword**

  Owner has been wrapped into a payable type to transfer the available fuse balance in the contract to the owner with low level **call.** However, **payable** is not required and can be removed.



```
    function withdrawFuse() public override onlyOwner {
        (bool sent, ) = payable(owner()).call{value: address(this).balance}("");
        require(sent, "Failed to send FUSE");
    }
```

- **Unindexed critical event parameters.**
  [#L46]event PurchaseLimitChanged doesn't have any indexed parameters. Unindexed parameters makes it difficult to track important data for off-chain monitoring tools.



```
    event TokenPurchase(
        address indexed purchaser,
        uint256 fuseAmount,
        uint256 tokenAmount
    );

    event PurchaseLimitChanged(uint256 oldPurchaseLimit, uint256 newPurchaseLimit);

```

Recommendation: Consider indexing event parameters to avoid the task of off-chain services searching and filtering for specific events.

Closing Summary:

Several issues of High, Medium and Low severity were found. Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the **FuseFi**. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the FuseFi team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.