#HashEx
BLOCKCHAIN SECURITY

PurpleSale Lock

smart contracts
final audit report

May 2024

hashex.org

contact@hashex.org

Contents

1. Disclaimer 3
2. Overview 4
3. Found issues 6
4. Contracts 8
5. Conclusion 23
Appendix A. Issues' severity classification 24
Appendix B. Issue status description 25
Appendix C. List of examined issue types 26
Appendix D. Centralization risks classification 27

1. Disclaimer

2. Overview

HashEx was commissioned by the PurpleSale team to perform an audit of their smart contract. The audit was conducted between 03/04/2024 and 04/04/2024.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available in the Binance Smart Chain testnet at 0x7c1Dbc796Db2B27453E5C739Cd9a24a05882639f.

Update. The PurpleSale team has responded to this report. The updated contracts are available in the Binance Smart Chain testnet at address 0x7E46a8Aa98fe51015D8F66533Fb04424f718E7FA.

Please note that no code verification for deployed contracts on mainnet was made. Deployed code may not be the same as what was audited. To ensure that the deployed contracts are the same as those audited, users must independently verify the SHA-1 hashes of the verified contract code.

2.1 Summary

Project name	PurpleSale Lock
URL	https://purplesale.finance
Platform	Binance Smart Chain
Language	Solidity
Centralization level	● Low
Centralization risk	● Low

2.2 Contracts

Name	Address
TokenTimeLock	
Interfaces and imported contracts	

3. Found issues

16
Total issues

●	Critical	4 (25%)
●	High	2 (13%)
●	Medium	3 (19%)
●	Low	4 (25%)
●	Info	3 (18%)

Cc9. TokenTimeLock

ID	Severity	Title	Status
Cc9I2e	● Critical	Incorrect indexation	✓ Resolved
Cc9I31	● Critical	Possible irrevocable funds locked with the launchLock function	✓ Resolved
Cc9I2a	● Critical	Unlocking amount can exceed locked amount	✓ Resolved
Cc9I99	● Critical	Broken vesting schedule updating	✓ Resolved
Cc9I30	● High	Lack of tests and documentation	✓ Acknowledged
Cc9I9a	● High	Wrong vesting amount calculation	✓ Resolved
Cc9I2d	● Medium	Gas limit problem	✓ Resolved
Cc9I2c	● Medium	Possible locked funds	✓ Partially fixed
Cc9I29	● Medium	Lack of input validation	✓ Resolved
Cc9I28	● Low	Gas optimizations	✓ Partially fixed

Cc9I24	● Low	Default visibility of state variables	✓ Resolved
Cc9I27	● Low	Lack of error messages	✓ Resolved
Cc9I25	● Low	Result of token transfer is not checked	✓ Resolved
Cc9I23	● Info	Lack of documentation (NatSpec)	✓ Resolved
Cc9I2f	● Info	Public function	✓ Acknowledged
Cc9I26	● Info	Limited ERC20 token support	✓ Acknowledged

4. Contracts

Cc9. TokenTimeLock

Overview

The TokenTimeLock is a permissionless locker and vesting contract, allowing any user to lock an arbitrary ERC20 tokens with a single unlocking date or with a vesting schedule split by equal time frames and amounts.

Issues

Cc9I2e Incorrect indexation ● Critical ✓ Resolved

Incorrect indexation in the `_vestingClaim()` function: `claimCycleCountPerID[msg.sender][id]` should be changed to `claimCycleCountPerID[msg.sender][index]` and `LockedTokens[msg.sender][id][claimCount]` to `LockedTokens[msg.sender][index][claimCount]`.

Otherwise, all user's vesting claims will result in external call to `address(0)` since `LockedTokens[msg.sender][id][claimCount].Token` is zeroed after the state `id` variable has been iterated.

```
function _vestingClaim(uint index) public {
    ...
    uint claimCount = claimCycleCountPerID[msg.sender][id] + 1;]
    address _token = LockedTokens[msg.sender][id][claimCount].Token;
    address _beneficiary = LockedTokens[msg.sender][id][claimCount].Beneficiary;
    ...
    IERC20Upgradeable(_token).safeTransfer(_beneficiary, claimmableAmount);
}
```

Recommendation

Fix the indexation to `[index]`.

Cc9I31 Possible irrevocable funds locked with the launchLock function

● Critical ✓ Resolved

The contract contains a `launchLock()` function designed to lock tokens for a user, with the lock owner's address being passed as a parameter. However, while the function is intended to create a lock for an owner, it inadvertently updates the parameter `cycleCountPerID` for `msg.sender` (the caller of the function) instead. Consequently, if a caller specifies an address different from their own as the intended owner, the locked funds will be irrevocable for the beneficiary.

```
function launchLock(address owner,  address token_, address beneficiary_, uint256
releaseTime_, uint256 amount_, bool _liquidity) external {
  ...
        Alllocked.push(LockedTokens[owner][id][1]);
        cycleCountPerID[msg.sender][id] = 1;

        allTokens[owner].push(token_);
        allAmount[owner].push(amount_);

        AllAmountLocked.push(amount_);
        AllTokensLocked.push(token_);
  ...
}

function Release(uint256 index) external {
        if (cycleCountPerID[msg.sender][index] == 1) {
            _normalClaim(index);
        } else {
            _vestingClaim(index);
        }
}

function _vestingClaim(uint index) public{
        require(claimCycleCountPerID[msg.sender][index] < cycleCountPerID[msg.sender]
[index], "Claim Complete");
    ...
}
```

Recommendation

Use the `owner` function parameter for the cycleCountPerID mapping instead of `msg.sender`.

Change

```
cycleCountPerID[msg.sender][id] = 1;
```

to

```
cycleCountPerID[owner][id] = 1;
```

Cc9I2a Unlocking amount can exceed locked amount ● Critical ✓ Resolved

The `Lock()` function uses dangerous check of all vesting schedule parts to be not less than 100%. The `Inputs.FirstPercent` is not checked and therefore can be set by the user to an arbitrary percent. Setting it more than 100% and disabling the `Inputs.Vesting` will result in unlocking amount greater than user's locked funds, i.e. stealing locked tokens of other users.

```
function Lock(inputs calldata Inputs) external {
    uint count = Inputs.cycleCount;
    uint totalPrecent = ((count - 1) * Inputs.cyclePercent) + Inputs.FirstPercent;
    require(totalPrecent >= 10000);
    ...
    IERC20Upgradeable(Inputs.Token).transferFrom(msg.sender,address(this),Inputs.amount);
    uint firstAmount = (Inputs.amount * Inputs.FirstPercent) / 10000;
    LockedTokens[msg.sender][id][1] = Locks({..., amount: firstAmount, ...});
    ...
}
```

Recommendation

Include safety check for the `Inputs.FirstPercent`.

Cc9I99 Broken vesting schedule updating ● Critical ✓ Resolved

The `changeReleaseTimeAndSpread()` function re-assembles user's vesting schedule with new percentages. If the updated number of cycles is less than the previous one, the `LockedTokens[]` vested amounts should be purged, but the `for()` loop iterates only once.

```
function changeReleaseTimeAndSpread(
    uint index,
    uint newTime,
    uint newCycleGap,
    uint[] memory newPercentages
) external {
    ...
    //Checks if the current counter is less than the old count, since the percentage has
been redistributed, the other indexes are deleted
    if (counter < count) {
        for (uint i = count; i <= count; i++) {
            delete LockedTokens[msg.sender][index][i];
        }
    }
}
```

Also, the release time is set to a wrong value in the middle of the function:

```
                if (count < counter) {
                    LockedTokens[msg.sender][index][counter] = Locks({
                        ...
                        releaseTime: LockedTokens[msg.sender][index][count].releaseTime,
                        ...
                    });
```

However, it is updated later to the correct one.

```
        //checks if token is vested, the applys new cycle gap time
        if (count > 1) {
            for (uint256 i = 2; i <= count; i++) {
                newTime += newCycleGap;

                LockedTokens[msg.sender][index][i].releaseTime = newTime;
            }
        }
```

Cc9I30 Lack of tests and documentation ● High ✓ Acknowledged

The project doesn't contain any tests and documentation. We urgently recommend increasing test coverage. We also suggest providing the documentation section.

Cc9I9a Wrong vesting amount calculation ● High ✓ Resolved

The lock function incorrectly calculates the lock amount if the passed lock amounts for vesting add up to more than the overall amount. For example, suppose a lock is created for 100 tokens with an initial claim of 90 tokens and a vesting schedule set to distribute 6 tokens four times. The first lock amount should be 90, the second 6, the third 4, and the subsequent amounts should be zero. However, the function incorrectly calculates the fourth and fifth amounts as 4 instead of zero. This error can lead to a situation where the user may not be able to withdraw his tokens at all if there are insufficient tokens on the contract, or he might withdraw more than he should.

```
    function Lock(inputs calldata Inputs) external {
        ...
        if (Inputs.Vesting) {
            for (uint256 i = 2; i <= count; i++) {
                maxPrecent += Inputs.cyclePercent;

                if (maxPrecent > 10000) {
                    maxPrecent -= Inputs.cyclePercent;
                    uint256 percent = 10000 - maxPrecent;
                    percentAmount = Inputs.amount * percent / 10000;
                }

                lastTime += Inputs.cyclereleaseTime;

                LockedTokens[msg.sender][id][i] = Locks({
                    owner: msg.sender,
                    Token: Inputs.Token,
                    Beneficiary: Inputs.Beneficiary,
                    amount: percentAmount,
                    releaseTime: lastTime,
                    liquidity: Inputs.liquidity,
                    Claimed: false
                });
            }
            ...
        }
```

Proof of concept

Foundry test

```
    function testVesting() public {
        token.approve(address(c), 100 ether);
        TokenTimeLockRecheck.inputs memory input = TokenTimeLockRecheck.inputs({
            Token: address(token),
            Beneficiary: alice,
            amount: 1 ether,
            liquidity: false,
            Vesting: true,
            FirstPercent: 9000,
            firstReleaseTime: 0,
            cyclePercent: 600,
            cyclereleaseTime: 0,
            cycleCount: 5
        });

        c.Lock(input);

        uint256 lockId = c.id();
        (,,, uint256 lockAmount1,,,) = c.LockedTokens(address(this), lockId, 1);
        console.log("lock amount 1", lockAmount1);

        (,,, uint256 lockAmount2,,,) = c.LockedTokens(address(this), lockId, 2);
        console.log("lock amount 2", lockAmount2);

        (,,, uint256 lockAmount3,,,) = c.LockedTokens(address(this), lockId, 3);
        console.log("lock amount 3", lockAmount3);

        (,,, uint256 lockAmount4,,,) = c.LockedTokens(address(this), lockId, 4);
        console.log("lock amount 4", lockAmount4);
        (,,, uint256 lockAmount5,,,) = c.LockedTokens(address(this), lockId, 5);
        console.log("lock amount 5", lockAmount5);
```

Test output

```
[PASS] testVesting() (gas: 712763)
Logs:
  lock amount 1 90000000000000000
  lock amount 2 60000000000000000
  lock amount 3 40000000000000000
  lock amount 4 40000000000000000
  lock amount 5 40000000000000000
```

Recommendation

Fix calculation by updating `maxPercent` in the if clause.

Cc9I2d Gas limit problem ● Medium ✓ Resolved

The `getEachLock()`, `getAllLockedDetailsInContract()`, `getAllTokensAndAmountInContract()` functions return full never purged arrays, inevitably resulting in these functions become non-invocable.

```
function getEachLock() external view returns (Locks[] memory) {
    return PersonalLocked;
}

function getAllLockedDetailsInContract()
```

```
    external
    view
    returns (Locks[] memory)
{
    return Alllocked;
}

function getAllTokensAndAmountInContract()
    external
    view
    returns (address[] memory, uint256[] memory)
{
    return (AllTokensLocked, AllAmountLocked);
}
```

Recommendation

Mark the `AllAmountLocked` and `AllTokensLocked` arrays with public visibility or introduce a getter function for an arbitrary index or index range.

Cc9I2c Possible locked funds ● Medium ✓ Partially fixed

If the `Inputs.Vesting` is false in the `Lock()` function, then `totalPrecent` is checked against 100% but only the first lock is applied. If the first lock is less than 100%, other deposited funds will be locked.

```
function Lock(inputs calldata Inputs) external {
    ...
    uint count = Inputs.cycleCount;
    uint totalPrecent = ((count - 1) * Inputs.cyclePercent) +
            Inputs.FirstPercent;
    require(totalPrecent >= 10000);
    IERC20Upgradeable(Inputs.Token).transferFrom(
        msg.sender,
        address(this),
        Inputs.amount
     );
    uint percentAmount = (Inputs.amount * Inputs.cyclePercent) / 10000;
    uint firstAmount = (Inputs.amount * Inputs.FirstPercent) / 10000;
    LockedTokens[msg.sender][id][1] = Locks({
        owner: msg.sender,
        id: id,
        Token: Inputs.Token,
        Beneficiary: Inputs.Beneficiary,
        amount: firstAmount,
        releaseTime: Inputs.firstReleaseTime,
        dateLocked: block.timestamp,
        liquidity: Inputs.liquidity,
        fromSale: false,
        Claimed: false
    });

    if (Inputs.Vesting) {
        for (uint i = 2; i <= count; i++) {
            ...
            LockedTokens[msg.sender][id][i] = Locks({...});
        }
    }
}
```

A small part of user's tokens remains locked due to division remainder in the `Lock()` function.

```
function Lock(inputs calldata Inputs) external {
    ...
    uint percentAmount = (Inputs.amount * Inputs.cyclePercent) / 10000;
    uint firstAmount = (Inputs.amount * Inputs.FirstPercent) / 10000;
    ...
}
```

Another possible lock is related to situation where `Inputs.firstReleaseTime` is much greater than `Inputs.cyclereleaseTime`, meaning that cycled vested amounts can't be claimed before the first one.

```
function Lock(inputs calldata Inputs) external {
    ...
    LockedTokens[msg.sender][id][1] = Locks({
        owner: msg.sender,
        Token: Inputs.Token,
        Beneficiary: Inputs.Beneficiary,
        amount: firstAmount,
        releaseTime: Inputs.firstReleaseTime,
        liquidity: Inputs.liquidity,
        Claimed: false
    });
    ...
    uint lastTime = block.timestamp;
    ...
    lastTime += Inputs.cyclereleaseTime;
    ...
    LockedTokens[msg.sender][id][i] = Locks({
        owner: msg.sender,
        Token: Inputs.Token,
        Beneficiary: Inputs.Beneficiary,
        amount: percentAmount,
        releaseTime: lastTime,
        liquidity: Inputs.liquidity,
        Claimed: false
    });
}
```

Recommendation

Include the `Inputs.Vesting` into the total percent calculation or remove it completely and use `bool vesting = Inputs.cycleCount > 1`.

Cc9I29 Lack of input validation ● Medium ✓ Resolved

The `Lock()` function's parameters are not validated.

The `Inputs.cycleCount` is not checked against 0, resulting in possible math underflowing.

The vesting schedule starts from the moment of lock transaction, ignoring the first lock amount and first release date. This may cause a delayed vesting release since for loop in the `_vestingClaim()` function will stop after the first iteration.

Recommendation

Carefully check the input data to eliminate risks of malfunction.

Cc9I28 Gas optimizations ● Low Partially fixed

1. The `Locks` and `inputs` structs should be optimized by storing `bool` packed with the `address` to fill 7 and 8 slots, respectively.

2. The `id` field could be removed from the `Locks` struct since it's stored in form of `LockedTokens[id].id=id`.

3. The `fromSale` and `dateLocked` fields are not checked anywhere in the contract and can presumably be emitted in event.

4. Multiple and unnecessary reads from storage in the `launchLock()` function: `id`, `LockedTokens[owner][id][1]` variables.

5. Multiple and unnecessary reads from storage in the `Lock()` function: `id`, `LockedTokens[owner][id][1]`, `LockedTokens[msg.sender][id][i]` variables.

6. Multiple reads from storage in the `_vestingClaim()` function: `cycleCountPerID[msg.sender][index]`, `claimCycleCountPerID[msg.sender][index]` variables.

7. The variable and mappings `LockingDetails[][]`, `personalLockedCount[]`, `individualLocks[]`, `AllAmountLocked[]`, `AllTokensLocked[]`, `Alllocked[]`, `PersonalLocked[]`, `allTokens[]`, and `allAmount[]` are not in use and store duplicated data.

8. The `dataStorage` address is not used in the contract. The `initialize()` function and inherited `Initializable` contract have unclear functionality.

Cc9I24 Default visibility of state variables ● Low ✓ Resolved

Several state variables have been declared without explicit visibility. In Solidity, the default visibility for state variables is `internal`. However, relying on default visibility can lead to misunderstandings and potential security vulnerabilities if not carefully considered and documented.

- `dataStorage` (address)
- `personalLockedCount` (mapping from address to uint256)
- `individualLocks` (mapping from address to array of Locks structs)
- `cycleCountPerID` (mapping from address to mapping from uint256 to uint256)
- `claimCycleCountPerID` (mapping from address to mapping from uint256 to uint256)
- `AllAmountLocked` (array of uint256)
- `AllTokensLocked` (array of address)
- `Alllocked` (array of Locks struct)
- `PersonalLocked` (array of Locks struct)

Cc9I27 Lack of error messages ● Low ✓ Resolved

There are no error messages in the `Lock()` and `Release()` functions, specifically in requirements for input amounts. Reverting without a reason is discouraged since it doesn't help user to choose right parameters and may provoke an unnecessary gas spending on obviously reverting transaction.

```
function Lock(inputs calldata Inputs) external {
    ...
    require(totalPrecent >= 10000);
}

function Release(uint index) external {
    if (cycleCountPerID[msg.sender][index] == 1) {
        _normalClaim(index);
    } else {
        _vestingClaim(index);
    }
}

function _normalClaim(uint index) internal {
    ...
    require(
        block.timestamp >
            LockedTokens[msg.sender][index][claimCount].releaseTime
    );
}
```

Cc9I25 Result of token transfer is not checked ● Low ✓ Resolved

The contract does not check the returned results of the ERC20 transfer function.

```
function Lock(inputs calldata Inputs) external {
    ...
    IERC20Upgradeable(Inputs.Token).transferFrom(
                msg.sender,
                address(this),
                Inputs.amount
            );
}

function Release(uint index) external {
    if (cycleCountPerID[msg.sender][index] == 1) {
        _normalClaim(index);
    } else {
        _vestingClaim(index);
    }
}

function _normalClaim(uint index) internal {
    ...
    IERC20Upgradeable(_token).transfer(_beneficiary, amount__);
}
```

The ERC20 token standard mandates that the token transfer function should return a boolean value indicating the success or failure of the token transfer. Typically, tokens are designed to always return true, with the transaction failing if the transfer is unsuccessful. However, it is considered best practice to robustly handle the return values to ensure reliability.

Additionally, it is important to note that some implementations of the ERC20 token do not adhere strictly to the ERC20 standard and might not return a boolean upon transfer. Consequently, to accommodate all scenarios, it is advisable to utilize a library that addresses these variations, such as OpenZeppelin's SafeERC20, which provides a more secure and standardized approach to handling ERC20 token transfers.

Recommendation

Use OpenZeppelin's SafeERC20 library to handle token transfers.

Cc9I23 Lack of documentation (NatSpec) ● Info ✓ Resolved

We recommend writing documentation using NatSpec Format. This would help in development, as well as simplify user interaction with the contract (including using the block explorer).

Cc9I2f Public function ● Info ✓ Acknowledged

The `_vestingClaim()` function has public visibility. Without documentation, it's impossible to ensure that's intended but the paired `_normalClaim()` function is `internal`.

Cc9I26 Limited ERC20 token support ● Info ✓ Acknowledged

Tokens with transfer fees and rebasing tokens are not supported.

Locking tokens with transfer fees may result in last users with active locks of this unsupported token being unable to unlock.

Locked rebasing tokens may result in math problems: all excesses will remain locked and all negative rebasing will result in unlocking failures.

Recommendation

Enrich the documentation to inform users that they should not use unsupported tokens.

Cca. Interfaces and imported contracts

Overview

The contracts ReentrancyGuard, AddressUpgradeable, SafeERC20Upgradeable, IERC20Upgradeable, IERC20PermitUpgradeable are imported from the OpenZeppelin repository without modifications.

Idatabase is a minimal interface with only 3 view functions for an external contract that is not used in the audited TokenTimeLock.

5. Conclusion

4 critical, 2 high, 3 medium, 4 low severity issues were found during the audit. 4 critical, 1 high, 2 medium, 3 low issues were resolved in the update.

This audit includes recommendations on code improvement and the prevention of potential attacks.

Please note that no code verification for deployed contracts on mainnet was made. Deployed code may not be the same as what was audited. To ensure that the deployed contracts are the same as those audited, users must independently verify the SHA-1 hashes of the verified contract code.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. Issue status description

✓ **Resolved.** The issue has been completely fixed.

✓ **Partially fixed.** Parts of the issue have been fixed but the issue is not completely resolved.

✓ **Acknowledged.** The team has been notified of the issue, no action has been taken.

? **Open.** The issue remains unresolved.

Appendix C. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

Appendix D. Centralization risks classification

Centralization level

- 🔴 **High.** The project owners can manipulate user's funds, lock user's funds on their will (reversible or irreversible), or maliciously update contracts parameters or bytecode.
- 🟠 **Medium.** The project owners can modify contract's parameters to break some functions of the project contract or contracts, but user's funds remain withdrawable.
- 🟢 **Low.** The contract is trustless or its governance functions are safe against a malicious owner.

Centralization risk

- 🔴 **High.** Lost ownership over the project contract or contracts may result in user's losses. Contract's ownership belongs to EOA or EOAs, and their security model is unknown or out of scope.
- 🟠 **Medium.** Contract's ownership is transferred to a contract with not industry-accepted parameters, or to a contract without an audit. Also includes EOA with a documented security model, which is out of scope.
- 🟢 **Low.** Contract's ownership is transferred to a well-known or audited contract with industry-accepted parameters.

contact@hashex.org

@hashex_manager

blog.hashex.org

linkedin

github

twitter

HashEx

BLOCKCHAIN SECURITY