DEFIMOON

be secure

Smart Contract Audit Report

July, 2023

DEFIMOON PROJECT

Audit and
Development

CONTACTS

defimoon.org
audit@defimoon.org
defimoon_org
defimoonorg
defimoon
defimoonorg

DEFIMOON

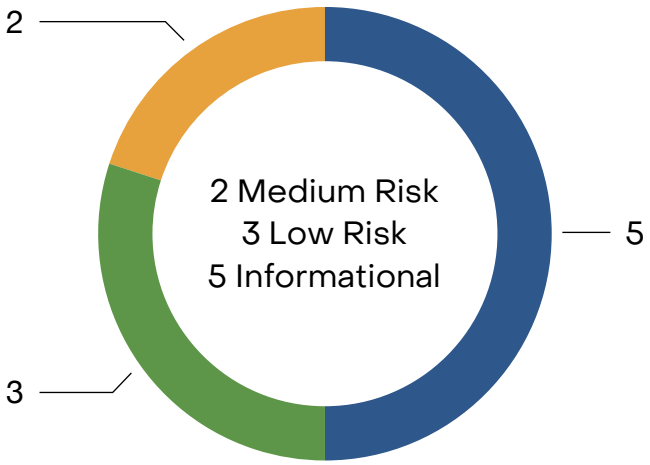
be secure

13 July 2023

This audit report was prepared by DefiMoon for Horizon-Sale.

Audit information

Description	Token sale contracts
Audited files	PublicSaleOverflow.sol, PrivateSaleOverflow.sol
Timeline	11 July 2023 – 13 July 2023
Audited by	Ilya Vaganov
Approved by	Artur Makhnach, Kirill Minyaev
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Source code	https://github.com/Horizon-Dex/token-sale/tree/d23c158b9cd5e615bd168bb6ca8d802be612310b
Network	Linea
Site	https://horizondex.io
Status	Not Passed



	High Risk	A fatal vulnerability that can cause the loss of all Tokens / Funds.
	Medium Risk	A vulnerability that can cause the loss of some Tokens / Funds.
	Low Risk	A vulnerability which can cause the loss of protocol functionality.
	Informational	Non-security issues such as functionality, style, and convention.

Disclaimer

This audit is not financial, investment, or any other kind of advice and could be used for informational purposes only. This report is not a substitute for doing your own research and due diligence should always be paid in full to any project. Defimoon is not responsible or liable for any loss, damage, or otherwise caused by reliance on this report for any purpose. Defimoon has based this audit report solely on the information provided by the audited party and on facts that existed before or during the audit being conducted. Defimoon is not responsible for any outcome, including changes done to the contract/contracts after the audit was published. This audit is fully objective and only discerns what the contract is saying without adding any opinion to it. Defimoon has no connection to the project other than the conduction of this audit and has no obligations other than to publish an objective report. Defimoon will always publish its findings regardless of the outcome of the findings. The audit only covers the subject areas detailed in this report and unless specifically stated, nothing else has been audited. Defimoon assumes that the provided information and materials were not altered, suppressed, or misleading. This report is published by Defimoon, and Defimoon has sole ownership of this report. Use of this report for any reason other than for informational purposes on the subjects reviewed in this report including the use of any part of this report is prohibited without the express written consent of Defimoon. In instances where an auditor or team member has a personal connection with the audited project, that auditor or team member will be excluded from viewing or impacting any internal communication regarding the specific audit.

Audit Information

Defimoon utilizes both manual and automated auditing approach to cover the most ground possible. We begin with generic static analysis automated tools to quickly assess the overall state of the contract. We then move to a comprehensive manual code analysis, which enables us to find security flaws that automated tools would miss. Finally, we conduct an extensive unit testing to make sure contract behaves as expected under stress conditions.

In our decision making process we rely on finding located via the manual code inspection and testing. If an automated tool raises a possible vulnerability, we always investigate it further manually to make a final verdict. All our tests are run in a special test environment which matches the "real world" situations and we utilize exact copies of the published or provided contracts.

While conducting the audit, the Defimoon security team uses best practices to ensure that the reviewed contracts are thoroughly examined against all angles of attack. This is done by evaluating the codebase and whether it gives rise to significant risks. During the audit, Defimoon assesses the risks and assigns a risk level to each section together with an explanatory comment.

Audit overview

Major vulnerabilities have been found.

Contracts have major vulnerabilities related to integer operations and condition boundary crossings, as well as some minor issues or code that could be reused to improve its quality.

We recommend that you pay more attention to conditions and mathematical calculations, as well as consider the behavior of the contract in non-standard or unintended situations in order to prevent potential problems and vulnerabilities.

The contract contains events for most of the main functions, uses immutable for optimization, and also uses custom errors, which are great solutions. But we want to remind you that using custom errors has the opposite format for describing conditions as using require(), so you need to be extremely careful to avoid errors when defining condition boundaries.

In general, contracts are written following most of the Solidity development best practices and use proven contracts from OpenZeppelin. We also recommend looking into using OpenZeppelin's Address::sendValue() (source) as a function to send ETH.

Summary of findings

ID	Description	Severity
DFM-1	Integer overflow	Medium Risk
DFM-2	Incorrect Conditions	Medium Risk
DFM-3	Potential loss of owner	Low Risk
DFM-4	Lack of address check	Low Risk
DFM-5	Re-changing the time	Low Risk
DFM-6	Redundant receive function	Informational
DFM-7	Simplifying the claim function	Informational
DFM-8	Too low minimum value	Informational
DFM-9	Additional checks	Informational
DFM-10	Using non-strict bounds	Informational

Application security checklist

Compiler errors	Passed
Possible delays in data delivery	Passed
Timestamp dependence	Passed
Integer Overflow and Underflow	Not Passed
Race Conditions and Reentrancy	Passed
DoS with Revert	Passed
DoS with block gas limit	Passed
Methods execution permissions	Passed
Private user data leaks	Passed
Malicious Events Log	Passed
Scoping and Declarations	Passed
Uninitialized storage pointers	Passed
Arithmetic accuracy	Passed
Design Logic	Passed
Cross-function race conditions	Passed

Detailed Audit Information

Contract Programming

Solidity version not specified	Passed
Solidity version too old	Passed
Integer overflow/underflow	Not Passed
Function input parameters lack of check	Not Passed
Function input parameters check bypass	Passed
Function access control lacks management	Passed
Critical operation lacks event log	Passed
Human/contract checks bypass	Passed
Random number generation/use vulnerability	Passed
Fallback function misuse	Passed
Race condition	Passed
Logical vulnerability	Not Passed
Other programming issues	Passed

Code Specification

Visibility not explicitly declared	Passed
Variable storage location not explicitly declared	Passed
Use keywords/functions to be deprecated	Passed
Other code specification issues	Passed

Gas Optimization

Assert () misuse	Passed
High consumption ‘for/while’ loop	Passed
High consumption ‘storage’ storage	Passed
“Out of Gas” Attack	Passed

Findings

DFM-1 «Integer overflow» <> PublicSaleOverflow <> PrivateSaleOverflow

Severity: Medium Risk

Description: In the PublicSaleOverflow and PrivateSaleOverflow contracts, when calculating ethersToRefund, the value may be rounded up, which can lead to contract performance issues.

This is because ethersToRefund is calculated as:

```
uint256 ethersToRefund = commitments[account] - ethersToSpend;
```

And the ethersToSpend variable can be calculated as:

```
uint256 ethersToSpend = (commitments[account] * ethersToRaise) /
totalCommitments;
```

If the multiplication result is not evenly divisible by totalCommitments, then rounding down will occur. This means that the result of the ethersToRefund calculation will be rounded up.

Recommendation: So in order for the value of ethersToRefund to be rounded down, you must first calculate it by dividing. For example, you can use the following code:

```
function simulateClaim(
        address account
) public view returns (uint256, uint256) {
    if (commitments[account] == 0) return (0, 0);

    if (totalCommitments >= refundThreshold) {

        uint256 ethersToRefund;

        if (ethersToRaise < totalCommitments) {
            uint256 ethersToRefund = commitments[account] * (totalCommitments -
ethersToRaise) / totalCommitments;
        }

        uint256 ethersToSpend = commitments[account] - ethersToRefund;
        uint256 tokensToReceive = (tokensToSell * ethersToSpend) /
ethersToRaise;

        return (ethersToRefund, tokensToReceive);
    } else {
        return (commitments[msg.sender], 0);
    }
}
```

In this case, you can refuse to use hardcoded values in the PrivateSaleOverflow contract and use the PublicSaleOverflow contract logic in the finish() function.

Also, the current implementation of the PrivateSaleOverflow::finish() function uses address(this).balance, however this may cause the contract to work incorrectly.

Because the PrivateSaleOverflow::claim(), PrivateSaleOverflow::overflowRefund() and PrivateSaleOverflow::finish() functions can all be called in the same time frame, the contract balance can change dynamically. Try to avoid using address(this).balance instead of a variable, as address(this).balance can change over time due to sending funds to users, receiving funds via receive(), or receiving funds due to a selfdestruct() call.

DFM-2 «Incorrect Conditions» <> PublicSaleOverflow

Severity: Medium Risk

Description: The functions PublicSaleOverflow::commit(), PublicSaleOverflow::claim(), and PublicSaleOverflow::finish() use the block.timestamp time check, and the contract logic assumes that PublicSaleOverflow::commit() can only be called up to the endTime, and PublicSaleOverflow::claim( ) and PublicSaleOverflow::finish() can only be called after the endTime. Only in this case the logic of the contract will work correctly.

However, in the current implementation it is possible to call the functions in a different order, such as calling PublicSaleOverflow::finish() or PublicSaleOverflow::claim() first and then calling PublicSaleOverflow::commit().
This is because these functions have a common valid point of operation at endTime.

```
function setTime(uint256 _startTime, uint256 _endTime) external onlyOwner {
    if (_startTime < block.timestamp) revert InvalidParam();
    if (_endTime < _startTime) revert InvalidParam();
    // available when _endTime >= _startTime
    // ...

function commit() external payable nonReentrant {
    if (
        !started ||
        block.timestamp <= startTime ||
        block.timestamp > endTime
    ) revert NotStartedOrAlreadyEnded();
    // available when endTime >= block.timestamp
    // ...

function claim() external nonReentrant returns (uint256, uint256) {
    if (block.timestamp < endTime) revert NotStartedOrAlreadyEnded();
    // available when endTime <= block.timestamp
    // ...

function finish() external onlyOwner {
    if (block.timestamp < endTime) revert NotFinished();
    // available when endTime <= block.timestamp
    // ...
```

As you can see, the conditions have >= or <= constraints, resulting in a total valid endTime value.

Also, the two variables startTime and endTime can all be set to the same value, which is not expected by the contract logic.

Recommendation: This vulnerability can lead to unexpected consequences and incorrect operation of the contract, which was not foreseen and is not processed in any way.

The best practice would be to implement the conditions in such a way that they do not have a common interval.

DFM-2 «Incorrect Conditions» <> PrivateSaleOverflow

Severity: Medium Risk

Description: The functions PrivateSaleOverflow::commit(), PrivateSaleOverflow::claim(), PrivateSaleOverflow::overflowRefund() and PrivateSaleOverflow::finish() use the block.timestamp time check, and the contract logic assumes that PrivateSaleOverflow::commit() can only be called up to the refundEndTime, PrivateSaleOverflow::overflowRefund() and PrivateSaleOverflow::finish() can only be called after the refundEndTime, and PrivateSaleOverflow::claim( ) can only be called after the claimEndTime. Only in this case the logic of the contract will work correctly.

However, in the current implementation it is possible to call the functions in a different order, such as calling PrivateSaleOverflow::finish() or PrivateSaleOverflow::overflowRefund() first and then calling PrivateSaleOverflow::commit().
This is because these functions have a common valid point of operation at refundEndTime.

```
function setTime(
    uint256 _startTime,
    uint256 _refundEndTime,
    uint256 _claimEndTime
) external onlyOwner {
    if (_startTime < block.timestamp) revert InvalidParam();
    if (_refundEndTime < _startTime) revert InvalidParam();
    if (_claimEndTime < _refundEndTime) revert InvalidParam();
    // available when _refundEndTime >= _startTime
    // available when _claimEndTime >= _refundEndTime
    // ...


function commit(
    bytes32[] calldata _merkleProof
) external payable nonReentrant {
    // ...
    if (
        !started ||
        block.timestamp <= startTime ||
        block.timestamp > refundEndTime
    ) revert NotStartedOrAlreadyEnded();
    // available when refundEndTime >= block.timestamp
    // ...


function overflowRefund() external nonReentrant returns (uint256) {
    if (block.timestamp < refundEndTime) revert NotStartedOrAlreadyEnded();
    // available when refundEndTime <= block.timestamp
    // ...

function finish() external onlyOwner returns (uint, uint) {
    if (block.timestamp < refundEndTime) revert NotFinished();
    // available when refundEndTime <= block.timestamp
    // ...

function claim() external nonReentrant returns (uint256) {
    if (block.timestamp < claimEndTime) revert NotStartedOrAlreadyEnded();
    // available when claimEndTime <= block.timestamp
    // ...
```

As you can see, the conditions have >= or <= constraints, resulting in a total valid refundEndTime value.

In addition, all three variables startTime, refundEndTime and claimEndTime can be set to the same value, which is not expected by the contract logic.

Recommendation: This vulnerability can lead to unexpected consequences and incorrect operation of the contract, which was not foreseen and is not processed in any way.

The best practice would be to implement the conditions in such a way that they do not have a common interval.

Also, the variables refundEndTime and claimEndTime are incorrectly named because they reflect the start time of the stage, not the end time. It's better to change them to "refundStartTime" and "claimStartTime".

DFM-3 «Potential loss of owner» <> PublicSaleOverflow <> PrivateSaleOverflow

Severity: Low Risk

Description: The PublicSaleOverflow and PrivateSaleOverflow contracts inherit the Ownable contract from OpenZeppelin which includes the renounceOwnership function. This function resets the owner of the contract without the possibility of restoring it, which can lead to irreparable consequences if this function is called, since most of the functionality of contracts is available only to the owner.

Also, the Ownable::transferOwnership function is not safe either.

Recommendation: Most of the functions in your contract require owner permissions, and as a result, loss of permissions can become critical. The best solution would be to stop using OpenZeppelin's renounceOwnership function. For example, like this:

```
function renounceOwnership() public override onlyOwner {
    revert("Renounce ownership disabled");
}
```

It's also best practice to use transfer the owner in two steps, like this.

DFM-4 «Lack of address check» <> PublicSaleOverflow <> PrivateSaleOverflow

Severity: Low Risk

Description: The PublicSaleOverflow::finish() and PrivateSaleOverflow::finish() functions can send tokens to burnAddress. However, burnAddress is not checked in any way in the contract constructor.

If you specify address(0) as the burnAddress in the current implementation, then if you send the rest of the tokens, a revert will almost certainly occur. This is because most ERC20 contracts (including the OpenZeppelin implementation) forbid token transfers to address(0). In this case, it will never be possible to call the PublicSaleOverflow::finish() or PrivateSaleOverflow::finish() functions and receive the collected ETH.

Recommendation: We recommend adding a check in the constructor that burnAddress != address(0) to avoid these problems.

In addition, the name burnAddress does not match the purpose of the address, as it is not guaranteed to be a burn address and can be used just to send the rest of the tokens.

DFM-5 «Re-changing the time» <> PublicSaleOverflow

Severity: Low Risk

Description: The PublicSaleOverflow::setTime() function can be called any number of times, even when sales have started.Since all the main functions of the contract depend on startTime and endTime, manipulation of this function can change the behavior of the contract.

In addition, this function cannot guarantee users that the conditions will not be changed and that they will receive their tokens on time.

Also, the PublicSaleOverflow::start() function can be called before the time has been set, which can be confusing.

Recommendation: The best practice to solve these problems is to set the time in the PublicSaleOverflow::start() function and abandon the PublicSaleOverflow::setTime() function. For example, like this:

```
function start(uint256 _startTime, uint256 _endTime) external onlyOwner {
    if (endTime > 0) revert AlreadyStarted();
    if (_startTime < block.timestamp) revert InvalidParam();
    if (_endTime < _startTime) revert InvalidParam();

    startTime = _startTime;
    endTime = _endTime;

    salesToken.safeTransferFrom(msg.sender, address(this), tokensToSell);
}
```

DFM-5 «Re-changing the time» <> PrivateSaleOverflow

Severity: Low Risk

Description: The PrivateSaleOverflow::setTime() function can be called any number of times, even when sales have started.Since all the main functions of the contract depend on startTime, refundEndTime and claimEndTime, manipulation of this function can change the behavior of the contract.

In addition, this function cannot guarantee users that the conditions will not be changed and that they will receive their tokens on time.

Also, the PrivateSaleOverflow::start() function can be called before the time has been set, which can be confusing.

Recommendation: The best practice to solve these problems is to set the time in the PrivateSaleOverflow::start() function and abandon the PrivateSaleOverflow::setTime() function. For example, like this:

```
function start(
    uint256 _startTime,
    uint256 _refundEndTime,
    uint256 _claimEndTime
) external onlyOwner {
    if (_claimEndTime > 0) revert AlreadyStarted();
    if (_startTime < block.timestamp) revert InvalidParam();
    if (_refundEndTime < _startTime) revert InvalidParam();
    if (_claimEndTime < _refundEndTime) revert InvalidParam();

    startTime = _startTime;
    refundEndTime = _refundEndTime;
    claimEndTime = _claimEndTime;

    salesToken.safeTransferFrom(msg.sender, address(this), tokensToSell);
}
```

DFM-6 «Redundant receive function» <> PublicSaleOverflow <> PrivateSaleOverflow

Severity: Informational

Description: The PublicSaleOverflow::receive() and PrivateSaleOverflow::receive() functions are redundant as no functionality requires the ability to send ETH outside of the contract functions.

In addition, ETH sent via receive() will not be able to be withdrawn, which may result in loss of funds.

Recommendation: The best practice is to avoid using the receive() function unless absolutely necessary.

DFM-7 «Simplifying the claim function» <> PublicSaleOverflow <> PrivateSaleOverflow

Severity: Informational

Description: The PublicSaleOverflow::claim() function uses calculations similar to those in the PublicSaleOverflow::simulateClaim() function.

Using the same code makes it harder to read, understand, and change the code.

Recommendation: We recommend using ready-made functionality to improve the quality of the code. For example, you can change the PublicSaleOverflow::simulateClaim() function to be public and use it in the PublicSaleOverflow::claim() function, like this:

```
function claim() external nonReentrant returns (uint256 ethersToRefund, uint256
tokensToReceive) {
    if (block.timestamp <= endTime) revert NotStartedOrAlreadyEnded(); // fixed

    if (commitments[msg.sender] == 0) revert InsufficientCommitment();

    if (userClaimed[msg.sender] == true) revert HasClaimed();

    (ethersToRefund, tokensToReceive) = simulateClaim(msg.sender);
    userClaimed[msg.sender] = true;

    if (totalCommitments < refundThreshold) commitments[msg.sender] = 0;

    if (tokensToReceive > 0) {
        salesToken.safeTransfer(msg.sender, tokensToReceive);
        emit ClaimTokens(msg.sender, tokensToReceive);
    }

    if (ethersToRefund > 0) {
        (bool success, ) = msg.sender.call{value: ethersToRefund}("");
        require(success, "Failed to transfer ether");
        emit ClaimETH(msg.sender, ethersToRefund);
    }
}
```

The PrivateSaleOverflow::claim() function can be modified in a similar way.

DFM-8 «Too low minimum value» <> PublicSaleOverflow

Severity: Informational

Description: The PublicSaleOverflow::MIN_COMMITMENT variable is set to 1 (1 wei), which is the smallest unit of ETH (1e18 wei). Using very small values in calculations, particularly when dividing by other numbers, can result in 0 because Solidity only works with integers and always rounds down.

Users who commit with very little ETH will likely neither receive tokens nor get their ETH back.

Recommendation: We recommend setting PublicSaleOverflow::MIN_COMMITMENT to a higher value.

DFM-9 «Additional checks» <> PublicSaleOverflow <> PrivateSaleOverflow

Severity: Informational

Description: The constructor of the PublicSaleOverflow contract does not check whether the _salesToken variables are not 0.
The PrivateSaleOverflow contract constructor does not check the _salesToken and _root variables for difference from 0.

If by mistake these values are not specified, then the contract will have to be deployed again, spending extra gas.

Recommendation: We recommend adding additional checks.

DFM-10 «Using non-strict bounds» <> PrivateSaleOverflow

Severity: Informational

Description: The PrivateSaleOverflow::overflowRefund() function uses loose bounds in the condition

```
if (totalCommitments < ethersToRaise) revert NotOverflow();
```

However, if totalCommitments == ethersToRaise, then there will be no refunds either, and extra calculation operations will still be performed, spending extra gas.

Recommendation: A better solution would be to use a non-strict border in the condition like this:

```
if (totalCommitments <= ethersToRaise) revert NotOverflow();
```

Automated Analyses

Slither

Slither's automatic analysis not found vulnerabilities, or these false positives results .

Methodology

Manual Code Review

We prefer to work with a transparent process and make our reviews a collaborative effort. The goal of our security audits is to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high-level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, review open issue tickets, and investigate details other than the implementation.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system to make a final decision.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Appendix A — Finding Statuses

Resolved	Contracts were modified to permanently resolve the finding
Mitigated	The finding was resolved by other methods such as revoking contract ownership or updating the code to minimize the effect of the finding
Acknowledged	Project team is made aware of the finding
Open	The finding was not addressed