DEFIMOON

be secure

Smart Contract Audit Report

June, 2023

DEFIMOON PROJECT

Audit and
Development

CONTACTS

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

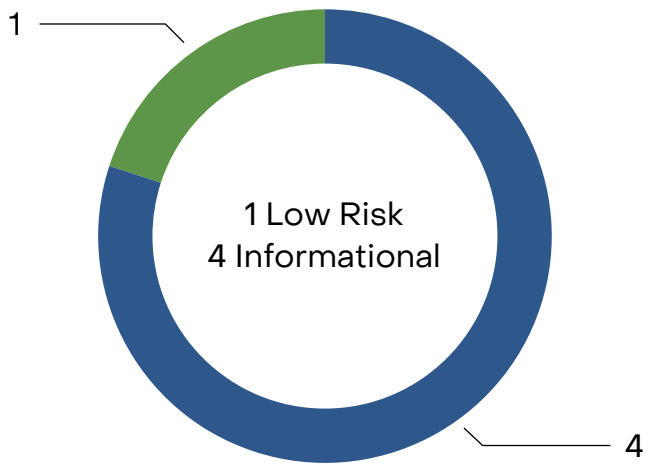
DEFIMOON
be secure

22 June 2023
This audit report was prepared by DefiMoon for GT-Protocol.

Audit information

Description	Tokens vesting contract, ERC20 tokens
Audited files	GTPToken.sol, GTPTokenBase.sol, GTPVesting.sol
Timeline	21 June 2023 - 22 June 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/global-traders-protocol/global-traders-protocol/tree/0395ff4cb923fad6fb8627868fdc10acb0232c36
Site	gt-protocol.io
Status	Passed

Disclaimer



	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.

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

No critical vulnerabilities found.

In general, contracts are written quite well, there are no critical vulnerabilities and problems, but contracts can be modified and optimized. The best practice would be to change the logic of the Ownable contract to be safer and optimize all loops in the Vesting contract, as well as implement the removal of completed vestings.

In addition, we recommend adding more events, including for all the main functions of the Vesting contract. Events are a very important component - they help to collect statistics, track interactions with a contract programmatically, and also search by topics, including in the block explorer.

Summary of findings

ID	Description	Severity
DFM-1	Potential loss of owner	Low Risk
DFM-2	Setters only work one way	Informational
DFM-3	No error descriptions	Informational
DFM-4	Completed vestings are not deleted	Informational
DFM-5	Loops optimizations	Informational

Application security checklist

Compiler errors	Passed
Possible delays in data delivery	Passed
Timestamp dependence	Passed
Integer Overflow and Underflow	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	Passed
Function input parameters lack of check	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	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 «Potential loss of owner»

Severity: Low Risk

Description: The GTP, GTPBase and Vesting 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("Vesting: Renounce ownership disabled");
}
```

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

DFM-2 «Setters only work one way»

Severity: Informational

Description: Functions GTP::setAntisnipeDisable, GTP::setLiquidityRestrictorDisable, GTPBase::addToWhitelist and GTPBase::start are setter functions that change the contract store, however they only work in one direction, setting the values of the variables only to false or true. As a result, in case of an error or in an unforeseen situation, it will not be possible to change one or another flag to true or false.

Recommendation: Make sure you don't need setter functions under any circumstances, or add them if you might need them.

In addition, in order to simplify and optimize the code, you can refuse to use the antisnipeEnabled and liquidityRestrictionEnabled variables, and hence the GTP::setAntisnipeDisable and GTP::setLiquidityRestrictorDisable functions, since

```
address(liquidityRestrictor) != address(0)
```

and

```
address(antisnipe) != address(0)
```

checks are enough.

DFM-3 «No error descriptions»

Severity: Informational

Description: Functions GTP::_beforeTokenTransfer, GTP::setAntisnipeDisable, and GTP::setLiquidityRestrictorDisable use require but do not return an error message on failure, which can make it difficult to determine what exactly caused the error.

Recommendation: The best practice is to return an error message on failure wherever possible.

DFM-4 «Completed vestings are not deleted»

Severity: Informational

Description: After the vesting is completed, the user's vesting id is not removed from the vestingIds. As a result, it is impossible to easily and quickly find out if the user still has incomplete vestings, and also this approach is not optimal in terms of gas consumption, since the getAvailableBalance function is still called for completed vestings.

Recommendation: The best practice would be to remove the id of completed vestings from users. For example, like this:

```
function claim(address account) external {
    uint256 totalAmount;
    uint256 i = vestingIds[account].length;
    require(i > 0, "Not vested");

    for (i; i > 0; ) {
        unchecked { i--; }

        uint256 id = vestingIds[account][i];
        uint256 amount = getAvailableBalance(id);
        if (amount > 0) {
            totalAmount += amount;
            vestings[id].claimed += amount - vestings[id].unlocked;
            vestings[id].lastUpdate = block.timestamp;
            vestings[id].unlocked = 0;
        }
        if (vestings[id].claimed >= vestings[id].amount) {
            vestingIds[account][i] = vestingIds[account]
[vestingIds[account].length - 1];
            vestingIds[account].pop();
        }
    }
    gtp.safeTransfer(account, totalAmount);
}
```

DFM-5 «Loops optimizations»

Severity: Informational

The Vesting contract uses a large number of loops that can be greatly optimized for the gas to be used.

First, it's better to use unit256 instead of uint8 as i. It makes no sense to use uint8, since this variable is local and is not written to storage slots. In addition, the slot size in Solidity is 32 bytes, which means that using uint256 does not require any additional conversions.

Second, it's better to declare the constraint as a separate variable instead of using the .length method, which avoids having to get the length each time.

Thirdly, using unchecked for increment will save gas by ignoring built-in SafeMath checks.

We want to demonstrate the effectiveness of optimization with a small example. All function calls were independent and carried out on new contracts.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.11;

contract GasTest {

    uint256 private variable;
    uint256[] private arr;

    constructor() {
        arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    }

    // 83136 gas
    // function test() external {
    //     for (uint8 i = 0; i < arr.length; i++) {
    //         variable = arr[i];
    //     }
    // }

    // 82922 gas
    // function test() external {
    //     for (uint256 i = 0; i < arr.length; i++) {
    //         variable = arr[i];
    //     }
    // }

    // 81695 gas
    // function test() external {
    //     uint256 l = arr.length;
    //     for (uint256 i; i < l; i++) {
    //         variable = arr[i];
    //     }
    // }

    // 81485 gas
    // function test() external {
    //     for (uint256 i; i < arr.length; ) {
    //         variable = arr[i];
    //         unchecked { ++i; }
    //     }
    // }

    // 80258 gas
    // function test() external {
```

```
    //     uint256 l = arr.length;
    //     for (uint256 i; i < l; ) {
    //         variable = arr[i];
    //         unchecked { ++i; }
    //     }
    // }
}
```

This approach may slightly increase the cost of deploying the contract, but it will save a lot of gas when using functions, especially with a large number of iterations.

Also, the Vesting::holdTokens function can be further optimized:

```
function holdTokens(HoldParams[] calldata params) external onlyOwner {
    uint256 totalAmount;
    uint256 l = params.length;
    for (uint256 i; i < l; ) {
        _holdTokens(params[i]);
        totalAmount += params[i].amount;
        unchecked { ++i; }
    }
    gtp.safeTransferFrom(msg.sender, address(this), totalAmount);
    totalVest += totalAmount;
}
```

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