



Smart Contract Audit Report

January, 2023


ArtPad

DEFIMOON PROJECT

Audit and
Development


CONTACTS

<https://defimoon.org>
audit@defimoon.org

 [defimoon_org](#)

 [defimoonorg](#)

 [defimoon](#)

 [defimoonorg](#)

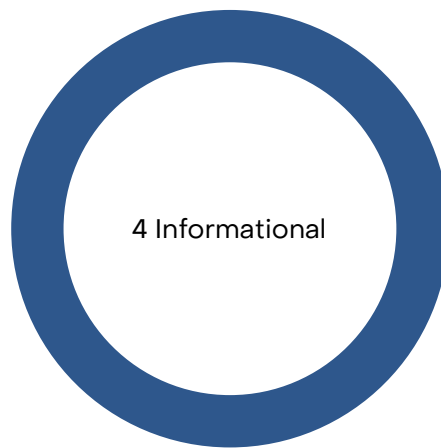


September 4th 2022 — January 4th 2023

This audit report was prepared by Defimoon for Artpad

Audit information

Description	The contract implements the ERC-20 type token and DEX interaction mechanics
Audited files	Artpad.sol, Staking.sol, wARTR.sol
Timeline	September 4th 2022 — January 4th 2023
Audited by	Ilya Vaganov
Approved by	Artur Makhnach, Cyrill Minyaev
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Source code	Audited from files provided by the client
Status	Passed



4

	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

No major security issues found after reaudit

In the `increaseAllowance` function, there is no check that the first argument (`owner`) is not the sender, which makes it possible for anyone to give permission to spend tokens owned by any other and, accordingly, withdraw all funds from that address (DFM-1).

Descriptions of functions do not comply with generally accepted standards called NatSpec, which impairs readability and understanding of the code (DFM-2).

Extra field in struct `User` - you can use the user's address instead of the `id` (DFM-3).

Redundant `SafeMath` library as it is used in contracts where the language version implements that functional by default (DFM-4).

Add a check on the amount of permission to spend user tokens by contract (DFM-5).

A custom contract has been written, the logic of which mimics the one already existing in the OZ library (DFM-6).

In contracts that inherit Ownable.sol access, the `_msgSender` function is a safer replacement for the classic `msg.sender` (DFM-7).

Double-Spend Exploit is also present in the current implementation (DFM-8).

Incorrect style of variable names you use is both camelCase and snake_case which makes it difficult to read the code (DFM-9).

In the logic of contracts, there are no events that are called in the main functions, which can complicate interaction with it (DFM-10).

You can change the condition in the space function to save more gas (DFM-11).

The `payable` modifier is placed where it is not necessary at all (DFM-12).

Summary of findings

According to the standard audit assessment, the audited solidity smart contracts are fairly secure but are not ready for production.

ID	Description	Severity	Status
<u>DFM-1</u>	Unlimited spending	High Risk	Resolved
<u>DFM-2</u>	Non-adherence to the "NatSpec" comment	Informational	Acknowledged
<u>DFM-3</u>	Redundant field in User struct	Informational	Acknowledged
<u>DFM-4</u>	Redundant library	Informational	Resolved
<u>DFM-5</u>	No approve check for transferFrom	Medium Risk	Resolved
<u>DFM-6</u>	Redundant logic	Informational	Acknowledged
<u>DFM-7</u>	Use of unsafe function	Low Risk	Resolved
<u>DFM-8</u>	Allowance Double-Spend Exploit	Informational	Resolved
<u>DFM-9</u>	Using snakeCase in variables and function	Informational	Acknowledged
<u>DFM-10</u>	No events in main functions	Informational	Resolved
<u>DFM-11</u>	Long require	Informational	Resolved
<u>DFM-12</u>	Unnecessary payable modifiers	Informational	Resolved

Launchpad Check list

Description	Status
No mint function found, owner cannot mint tokens after initial deploy	✓
Owner can't set max tx amount	✓
Owner can't set fees over 25%	✓
Owner can't pause trading[1]	✗
Owner can't blacklist wallets	✓

```
[1] function sellingActiveToggle() public onlyOwner {  
    sellingIsActive = !sellingIsActive;  
}
```

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 «Unlimited spending»

Severity: High Risk

Status: Resolved

Description:

The current implementation allows any user to allow spending **OTHER PEOPLE's** tokens without the permission of their owner.

In order to send someone else's tokens to yourself, you need to get permission from the owner using the **approve** method, but in the current implementation there is also the **increaseAllowance/decreaseAllowance** method, which allows anyone to get permission to spend a token from any other wallet.

Exploit Scenario:

1. The first user has a certain number of your tokens, the second user knows about it and wants to send them to himself.

```
28 console.log(
29   "Before balance owner: ",
30   ethers.utils.formatEther(await mainToken.balanceOf(owner.address))
31 );
32 console.log(
33   "Before balance user: ",
34   ethers.utils.formatEther(await mainToken.balanceOf(addr.address))
35 );
Before balance owner: 100000000.0
Before balance user: 0.0
```

2. The second user calls the increase Allowance method and passes 99% of all tokens of the first user there as the first parameter. Next, if we check how much he is allowed to spend, we will see the right number.

```
38 await mainToken
39   .connect(addr)
40   .increaseAllowance(
41     owner.address,
42     addr.address,
43     ethers.utils.parseEther("99999999")
44   );
Approve for: 99999999.0
```

3. Next, the second user calls the transfer From function and unhindered output the required number of tokens

```
54 try {
55     await mainToken
56       .connect(addr)
57       .transferFrom(
58         owner.address,
59         addr.address,
60         ethers.utils.parseEther("99999999"))
61   );
62 } catch (error) {
63   console.log(error);
64 }
65
66 console.log(
67   "After balance staking: ",
68   ethers.utils.formatEther(await mainToken.balanceOf(staking.address))
69 );
70 console.log(
71   "After balance user: ",
72   ethers.utils.formatEther(await mainToken.balanceOf(addr.address))
73 );
```

After balance staking: 0.0
After balance user: 99999999.0

Recommendation:

Do not pass the **owner** argument, but use **_msgSender** instead, which does not allow manipulating other people's tokens on behalf of another user.

DFM-2 «Non-adherence to the "NatSpec" comment format»

Severity: **Informational**

Status: **Acknowledged**

Description:

Documentation and commenting in the current contract is not standardized. It is not informative enough, which makes the code difficult to read. Most importantly, it also don't follow the semantic rules required for the web3 applications (blockchain explorers) to process contracts properly.

Recommendation:

It is recommended at least to add attributes in comments such as “@notice”, “@param”. You can read about it [here](#).

DFM-3 «Redundant field in User struct»

Severity: Informational

Status: Acknowledged

Description:

There is an extra field «id» in the User structure that can be replaced with an existing address field. This field makes interaction with methods where there is work with this structure more expensive for gas.

Recommendation:

Delete the current field «id» and interact with the user's address.

DFM-4 «Redundant library»

Severity: Informational

Status: Resolved

Description:

The SafeMath library is not relevant for the current version of solidity since the overflow problem was solved in solidity version 0.8.0. Its presence makes the smart contract deployment more expensive.

Recommendation: Remove using and import for SafeMath library.

DFM-5 «No approve check for transferFrom»

Severity: Medium Risk

Status: Resolved

Description:

There is no check that allows you to tell what restrictions a user has on spending tokens from another address.

Recommendation:

Add condition: `require(allowance(owner, spender) >= amount)`.

DFM-6 «Redundant logic»

Severity: [Informational](#)

Status: [Acknowledged](#)

Description:

The contract ERC20 that is written above the main one almost completely repeats the logic of the existing contract from the openzeppelin library, which also makes the smart contract deployment more expensive.

Recommendation:

If you still need to customize some methods, you can use [override](#) keyword before all the modifiers and describe the updated logic of the method in the body.

ReAudit-Note: Since your token contract is almost a complete copy of OpenZeppelin's ERC20 contract, it's always better to use a ready-made, proven solution. Thus, you could fully inherit the ERC20 contract.

Thus, it would suffice to add "import [«@openzeppelin/contracts/token/ERC20/ERC20.sol»;](#)" instead of a native implementation of the ERC20 contract.

The main difference is that you modify `wARTR::increaseAllowance` so that the staking contract can call this function. In that case, it would be enough for you to override this function to change its logic.

In addition, I would like to note that this approach, which implements `approve` instead of the user (the `StakingARTR::Stake` function), is doubtful and such an application should be avoided and the classic "approve -> transferFrom" approach should be used, in which `approve` is performed by the user himself. This approach is more secure for the user and is a double check, thanks to which the user understands exactly what is required of him, because he signs two specific transactions.

DFM-7 «Use of unsafe function»

Severity: Low Risk

Status: Resolved

Description:

In contracts that inherit Ownable.sol access, the `msgSender` function is a safer replacement for the classic `msg.sender`

Recommendation:

In contracts that inherit Ownable.sol, replace `msg.sender` with `msgSender`.

DFM-8 «Allowance Double-Spend Exploit»

Severity: [Informational](#)

Status: [Resolved](#)

Description: As it presently is constructed, the contract is vulnerable to the allowance double-spend exploit, as with other BEP20 tokens.

Exploit Scenario:

1. Alice allows Bob to transfer N amount of Alice's tokens ($N > 0$) by calling the `approve()` method on Token smart contract (passing Bob's address and N as method arguments)
2. After some time, Alice decides to change from N to M ($M > 0$) the number of Alice's tokens Bob is allowed to transfer, so she calls the `approve()` method again, this time passing Bob's address and M as method arguments.
3. Bob notices Alice's second transaction before it was mined and quickly sends another transaction that calls the `transferFrom()` method to transfer Alice's tokens somewhere.
4. If Bob's transaction will be executed before Alice's transaction, then Bob will successfully transfer N Alice's tokens and will gain an ability to transfer another M tokens.
5. Before Alice notices any irregularities, Bob calls `transferFrom()` method again, this time to transfer M Alice's tokens.

Recommendation: The exploit (as described above) is mitigated through use of function `safeApprove()`.

Pending community agreement on an ERC standard that would protect against this exploit, we recommend that developers of applications dependent on `approve()` / `transferFrom()` should keep in mind that they have to set allowance to 0 first and verify if it was used before setting the new value.

Addition: Renaming the function will not lead to any changes, meaning that you will import the contract from "`@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol`" and inherit from it and use the built-in `safeApprove` function.

DFM-9 «Using snakeCase in variables and function naming»

Severity: Informational

Status: Acknowledged

Description: The names of many variables are written in a "mixed" style, which complicates the reading of the code as it can confuse those who will read the code.

Recommendation: Rewrite all variable and function names into snakeCase style for easier code reading.

ReAudit-Note: snakeCase (camelCase or mixedCase) style is the main style of the Solidity language. It is always worth sticking to this format for writing code so that it conforms to standards and is more readable.

DFM-10 «No events in main functions»

Severity: Informational

Status: Resolved

Description: The main functions do not use events, this can lead to a complication of interaction with the contract on the sites.

Recommendation: It is desirable to write events for each function and write comments on them.

DFM-11 «Long require»

Severity: Informational

Status: Resolved

Description:

In the state function, the condition checking the month number can be shortened by 2 times, which will make it cheaper to perform the function.

Recommendation:

It is recommended to change it like so: `require(months == 1 || months % 3 == 0)`

ReAudit-Note: The find is no longer relevant because the application logic has been changed.

DFM-12 «Unnecessary payable modifiers»

Severity: Informational

Status: Resolved

Description:

In functions where it is not intended for the user to send native tokens to the network, you do not need to write a `payable` modifier.

Recommendation:

Remove `payable` modifiers in methods where `msg.value` is not used.

Automated Analyses

Slither

Slither has reported 237 findings. These results were either related to code from dependencies, false positives or have been integrated in the findings or best practices of this report.

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