QuillAudits

Audit Report
April, 2022

For

MULTIVERSEPAD

THE#1 METAVERSE IDO|IGO|INO LAUNCHPAD

Contents

Scope of Audit

The scope of this audit was to analyze and document the MultiversePad smart contract codebase for quality, security, and correctness.

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
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- 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 analysed 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 analysed, 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 behaviour 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
Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Issue Categories

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

Risk-level	Description
High	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	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	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 severity 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	0	0	0	0
Acknowledged	0	1	3	9
Closed	0	1	0	0

Introduction

During the period of **April 6, 2022 to April 12 , 2022** - QuillAudits Team performed a security audit for MultiversePad's smart contract.

The code for the audit was taken from the Auditee:
MultiversePad Contract Deployed at:
https://bscscan.com/address/0x64d2906391a82721bb24925fc16a3eff20c8756e#code

Issues Found – Code Review / Manual Testing

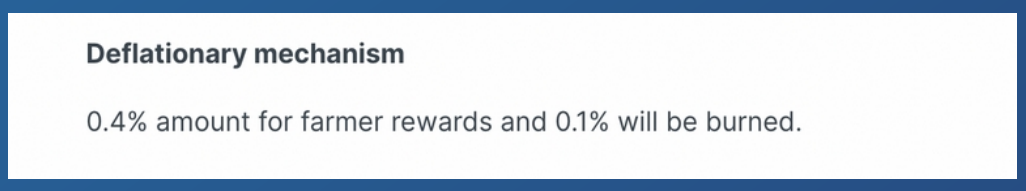
High severity issues

No issues found

Medium severity issues

1. Mathematical Error in fees calculation

According to the documentation, (https://docs.multiversepad.com) it is stated that the contract is supposed to work on a deflation mechanism and will charge 0.5% fee on every transaction that is not initiated or directed towards a whitelisted address. Out of that 0.5%, 0.4% amount is supposed to be distributed among farmers (address added in reward pool by the owner) and 0.1% is supposed to be burned.



However, according to the actual implementation, 0.45% is distributed among the farmers and 0.05% of fees is burned.

```
uint256 extractAmount = (amount * 5) / 1000;

uint256 burnAmount = (extractAmount * 10) / 100;
uint256 rewardAmount = (extractAmount * 90) / 100;
```

Here first the extract amount is 0.5%. Now burnAmount and rewardAmount are calculated from extractAmount. So the end, we are getting is 0.45% rewardAmount and 0.05% as burnAmount.

For instance, let's assume the amount is 100000. So extractAmount will be 0.5% of 100000 (i.e 500). BurnAmount will be 50 (which is 0.05% of the actual amount) and reward Amount will be 450 (which is 0.45% of the actual amount).

Recommendation
Currently, there is a 90:10 split of the extract amount which results in this unusual rewards distribution. If the team actually wants to achieve 0.4%, 0.1% split of extractAmount, it is recommended to perform a 80:20 split of the extractAmount. This can be achieved by, for burnAmount, multiply extractAmount by 20, and for rewardAmount, multiply extractAmount by 80. This way the team can make sure that reward distribution is happening as mentioned in the documentation (i.e 0.4%, 0.1%).

Status: Fixed
Auditor's Comment: The information provided in the whitepaper and the website about the reward distribution was incorrect. The MultiversePad team has acknowledged the same has made changes to the whitepaper and the official website.

2. NonWhiteListed addresses can make a transfer without paying 0.5% deflation fees

The token contract works on a deflationary mechanism which means a 0.5% fees will be charged on every transaction that is not initiated or directed towards the whitelisted address. Whitelisting of addresses is done via addWhitelistTransfer() which is only callable by the owner. The whitelist functionality is used to exempt exchanges etc from transfer fees. Any other user must pay a 0.5% fee to make a transfer. However, there is a function named transferWithoutDeflationary() whose visibility is public which means that any user can directly call transferWithoutDeflationary() and transfer his token to another address without paying 0.5% fee.

```
function transferWithoutDeflationary(address recipient, uint256 amount) public virtual override returns
        _transferWithoutDeflationary(_msgSender(), recipient, amount);
        return true;
    }
```

Recommendation
It is recommended to review the logic behind the calling of this function and enforce proper access control. In order to mitigate this issue, it is recommended to declare this function as internal.

Status: Acknowledged

Low severity issues

3. **Race conditions due to BEP20 Approve function.**

The standard ERC20 implementation contains a widely-known racing condition in its approve function, wherein a spender is able to witness the token owner broadcast a transaction altering their approval and quickly sign and broadcast ay transaction using transferFrom to move the current approved amount from the owner's balance to the spender. If the spender's transaction is validated before the owner's, the spender is able to spend their entire approval amount twice. As the BEP20 standard is proposed by deriving the ERC20 protocol of Ethereum, the race condition exists here as well.

Here is a possible attack scenario: Alice allows Bob to transfer N of Alice's tokens (N>0) by calling approve method on the Token smart contract passing Bob's address and N as method arguments 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 approve method again, this time passing Bob's address and M as method arguments Bobnotices Alice's second transaction before it was mined and quickly sends another transaction that calls·transferFrom· method to transfer N Alice's tokens somewhere If Bob's transaction will be executed before Alice's transaction, then Bob will successfully transfer N Alice's tokens and will gain the ability to transfer another M tokens Before Alice noticed that something went wrong, Bob calls ·transferFrom· method again, this time to transfer M Alice's tokens. So, Alice's attempt to change Bob's allowance from N to M (N>0 and M>0) made it possible for Bob to transfer N+M of Alice's tokens, while Alice never wanted to allow so many of her tokens to be transferred by Bob.

Recommendation
One possible solution to mitigate this race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards.

Reference
1. https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM
2. https://github.com/binance-chain/BEPs/blob/master/BEP20.md#5119-approve
3. https://eips.ethereum.org/EIPS/eip-20

Status: Acknowledged

4. **Not complying with BEP20 standard completely.**

BEP20 standard makes it mandatory for the tokens to define a function as getOwner, which should return the owner of the token contract.

5.1.1.6 getOwner

```
function getOwner() external view returns (address);
```

- Returns the bep20 token owner which is necessary for binding with bep2 token.
- **NOTE** - This is an extended method of EIP20. Tokens which don't implement this method will never flow across the Binance Chain and Binance Smart Chain.

However, the token contract doesn't implement/define any such function, as a result, the token may not flow across the Binance Chain and Binance Smart Chain, as stated by BEP20 interface documentation.

Recommendation
Consider adding the getOwner function.

Status: Acknowledged

5. **rewardPool[] Length problem: Transaction might fail if the length of rewardPool[] array is very large.**

MultiversePad contract has a function named countActiveRewardPool() which uses a for loop over an array. If the length of the rewardpool[] array is very large, This may lead to extreme gas costs up to the block gas limit and eventually fail the transaction.
In an extreme situation with a large number of rewardPool addresses, transaction gas may exceed the maximum block gas size and all transfers will be effectively blocked. If the owner's account gets compromised the attacker can make the token completely unusable for all users. The same issue also exists in _transfer function at #L-261.

```
function countActiveRewardPool() public  view returns (uint256){
    uint length=0;
 for(uint i=0;i<rewardPool.length;i++){
     if(mapRewardPool[rewardPool[i]].isActive){
         length++;
     }
 }
  return  length;
}
```

Exploit Scenario: Let's assume a scenario where a number of addresses in rewardpool is very large and by running a for loop over that length, the transaction fails due to out of gas issue. Now a user wants to transfer his token to another address. The user will call the transfer function, now if we look at the whole flow, _transfer() function will call countActiveRewardPool() function that has a for loop which consumes a large quantity of gas. So if countActiveRewardPool() fails, transfer will fail.

Recommendation
This issue can be avoided if the owner restricts the number of addresses added to the rewardpool array. It is also recommended to implement checks that prevent the owner from adding addresses to the rewardpool to a certain threshold.

Status: Acknowledged

Informational issues

6. Unlocked Pragma (>=0.6.0)

Contracts should be deployed with the same compiler version and flags that they have been tested 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 by removing the >= sign to lock the file onto a specific Solidity version. Moreover, Consider upgrading to a newer solidity version since this version is relatively old.

Status: Acknowledged

7. Inactive users are not popped out of rewardPool

The contract allows owner to add addresses to the reward pool and remove the users. Whenever the owner wants to remove a user from earning rewards, he can simply call removeRewardPool() and deactivate the user. This process, will only set the status as false and he won't receive the reward anymore. But the address is not actually popped out of rewardPool.

In contracts, there are several loops that iterate over the length of rewardPool. If deactivated addresses are removed from the rewardPool, it would slightly reduce the gas consumption. Moreover, if owner removes any address from earning rewards by mistake, there is no way to add that address back. The require statement of #L143 will fail and prevent the owner from adding that address back to the pool.

Recommendation
It is recommended to review the logic of the above-mentioned functions and implement a more optimized mechanism.

Status: Acknowledged

8. **Public functions that could be declared external in order to save gas.**

Whenever a function is not called internally, it is recommended to define them as external instead of public in order to save gas. For all the public functions, the input parameters are copied to memory automatically, and it costs gas. If your function is only called externally, then you should explicitly mark it as external. External function's parameters are not copied into memory but are read from calldata directly. This small optimization in your solidity code can save you a lot of gas when the function input parameters are huge.

Here is a list of function that could be declared external:
#L-141- addRewardPool()
#L-150- addWhitelistTransfer()
#L-158- removeWhitelistTransfer()
#L-168- removeRewardPool()
#L-184- getRewardPool()
#L-191- totalSupply()
#L-196- balanceOf()
#L-216- allowance()
#L-233- increaseAllowance()
#L-239- decreaseAllowance()
#L-360- setBeginDeflationFarming()
#L-365- getBeginDeflationary()
#L-398- name()
#L-403- symbol()
#L-408- decimals()
#L-425- withdrawErc20()

Status: Acknowledged

9. **Centralization Risk**

   Multiversepad (MTVP) is a token that mints the 100% of the total tokens liquidity to the deployer of the contract. The implementation is prone to centralization risk that may arise, if the deployer loses its private key.

   Recommendation
   Ensure the private keys of the owner account are diligently handled.

   Status: Acknowledged

10. **Using Block values as a proxy for time**

    Here in function _transfer() A control flow decision is made based on The 'block.timestamp' environment variable. Note that the values of variables like coinbase, gaslimit, block number, and timestamp are predictable and can be manipulated by malicious miners. Also, keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that the use of these variables introduces a certain level of trust into miners.

    Recommendation
    Developers should write smart contracts with the notion that block values are not precise, and the use of them can lead to unexpected effects. Alternatively, they may make use of oracles.

    Status: Acknowledged

11. **Stuck tokens in the contract gets transferred to the owner of the contract and the function can be invoked by anyone.**

    On #L425, there is a function named withdrawErc20() which is basically used to withdraw stuck tokens from the contract. If a user transfers any other tokens to this contract by mistake, this function can be called to rescue those token. This function will transfer those tokens to the owner of the contract. However, it is recommended to transfer those tokens to the sender. Moreover, the visibility of the function is set to public and anyone can call it. We don't see any practical usage of this. If possible, consider declaring this function as external and callable only by the owner. And also pass the _to address as a parameter. Basically, the owner must be able to call this function and transfer stuck tokens to the address passed as a function parameter.

The current implementation is not a threat from a security point of view, but it is recommended to review the logic of the function and make changes accordingly.

Status: Acknowledged

12. **Missing Explicit Visibility**

    On #L135 and L#136 the visibility of the mapping and array is not explicitly defined. Whenever visibility is not defined explicitly, default visibility is used. It is recommended to follow the best practice and explicitly define the required visibility.

    Status: Acknowledged

13. **Usage of unsafeMath**

    Since the contract relies on an older version of solidity (<0.8.0) there is a risk of integer underflow/overflow. #L-351,353,354,281 uses unsafe methods for arithmetic computations which is a bad practice and must be avoided.

    Recommendation

    It is recommended to use SafeMath library or upgrade to the latest version of solidity.

    Status: Acknowledged

14. **Gas optimization: For loop optimization**

    In countActiveRewardPool() and _transfer(), there is a for loop which iterates the value of rewardPool.length times. Each time the for loop executes, the value of rewardPool.length is computed which consumes some gas. This can be optimized by calculating the value of rewardPool.length outside the for loop. The optimized loop would look like this:

```
function countActiveRewardPool() public  view returns (uint256){
    uint length=0;
    uint poolLenght = rewardPool.length;
 for(uint i=0;i<poolLenght;i++){
     if(mapRewardPool[rewardPool[i]].isActive){
         length++;
     }
 }
  return  length;
}
```

Recommendation
Update the loop as recommended above.

Status: Acknowledged

Functional Tests

- Should test all getters	PASS
- OnlyOwner must be able to add an address to rewardPool	PASS
- OnlyOwner must be able to add address to whitelist transfer	PASS
- Should test removeWhitelistTransfer.	PASS
- Should test countActiveRewardPool,getRewardPool and removeRewardPool.	PASS
- Should test transfer	PASS
- Whitelisted address must be exempt from fees.	PASS
- Should test burn and burn from.	PASS
- Should test increaseallowance,decreaseallowance and transferFrom.	PASS
- Deflation fees must be correctly calculated and distributed among burn address and addresses in the reward pool.	PASS
- All non-whitelisted addresses must pay fees in order to transfer tokens.	FAIL

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the MultiversePad Smart Contract. We performed our audit according to the procedure described above.

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

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the MultiversePad platform. 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 MultiversePad Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

Audit Report
April, 2022

For

MULTIVERSEPAD

THE#1 METAVERSE IDO|IGO|INO LAUNCHPAD

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com