BLOCK SOLUTIONS

Smart Contract Code Review and Security Analysis Report for Meta Verse Market BEP20 Token Smart Contract

Request Date: 2022-02-10
Completion Date: 2022-02-12
Language: Solidity

Contents

Commission

Audited Project	Meta Verse Market BEP20 Token Smart Contract
Contract Address	0x6a3110CebA22Bc29FA14845afA30a69E5D574ab2
Contract Owner	0x45b1af9ae62416be2a3f494d5449ce3ae0c7d176
Contract Creator	0x8a911e1aff89a0a58e224da43e8e4d8a4d756614
Blockchain Platform	Binance Smart Chain Mainnet

Block Solutions was commissioned by META VERSE MARKET BEP20 TOKEN Smart Contract owners to perform an audit of their main smart contract. The purpose of the audit was to achieve the following:

- Ensure that the smart contract functions as intended.
- Identify potential security issues with the smart contract.

The information in this report should be used to understand the risk exposure of the smart contract, and as a guide to improve the security posture of the smart contract by remediating the issues that were identified.

META VERSE MARKET
BEP20 TOKEN Properties

Contract Token name	Meta Verse Market
Symbol	MVM
Decimals	9
Total Supply	9000000000
Developer Fee	1 %
Liquidity Fee	3 %
Marketing Fee	2 %
Tax Fee	3 %
Maximum Transaction Amount	90000000
Minimum Token Before Swap	9000
Developer Address	0x47d3550c1c17b77f3523a6d435fdd48cf0c6b75e
Marketing Address	0x7219c7602bf0633e26a723f2c7e43b58605bad7a
Contract Owner	0x45b1af9ae62416be2a3f494d5449ce3ae0c7d176
PancakeSwapV2Pair	0x61a979daba107e5f0a6117d9dd9a4cdd941068f0
PancakeSwapV2Router	0x10ed43c718714eb63d5aa57b78b54704e256024e

Contract Functions

Executables

i. function approve(address spender, uint256 amount) public override returns (bool)
ii. function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool)
iii. function deliver(uint256 tAmount) public
iv. function transfer(address recipient, uint256 amount) public override returns (bool)
v. function transferFrom(address sender, address recipient, uint256 amount) public override returns (bool)
vi. function unlock() public virtual

Owner Executable:

i. function afterPreSale() external onlyOwner
ii. function excludeFromFee(address account) public onlyOwner
iii. function excludeFromReward(address account) public onlyOwner()
iv. function includeInReward(address account) external onlyOwner()
v. function includeInFee(address account) public onlyOwner
vi. function lock(uint256 time) public virtual onlyOwner
vii. function prepareForPreSale() external onlyOwner
viii. function renounceOwnership() public virtual onlyOwners
ix. function setAllSaleFeesPercents(uint256 taxFee, uint256 liquidityFee, uint256 marketingFee, uint256 developerFee, uint256 burnFee) external onlyOwner
x. function setBurnFeePercent(uint256 burnFee) external onlyOwner()
xi. function setDeveloperFeePercent(uint256 developerFee) external onlyOwner()
xii. function setLiquidityFeePercent(uint256 liquidityFee) external onlyOwner()
xiii. function setMarketingFeePercent(uint256 marketingFee) external onlyOwner()
xiv. function setMarketingAddress(address _marketingAddress) external onlyOwner()
xv. function setMaxTxAmount(uint256 maxTxAmount) external onlyOwner()
xvi. function setNumTokensSellToAddToLiquidity(uint256 _minimumTokensBeforeSwap) external onlyOwner()
xvii. function setSwapAndLiquifyEnabled(bool _enabled) public onlyOwner
xviii. function setTaxFeePercent(uint256 taxFee) external onlyOwner()
xix. function transferOwnership(address newOwner) public virtual onlyOwner

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
Economy model of the contract.	Passed
Private user data leaks.	Passed
Malicious Events Log.	Passed
Scoping and Declarations.	Passed
Uninitialized storage pointers.	Passed
Arithmetic accuracy.	Passed
Design Logic.	Passed
Impact of the exchange rate.	Passed
Oracle Calls.	Passed
Cross-function race conditions.	Passed

Fallback function security.	Passed
Safe Open Zeppelin contracts and implementation usage.	Passed
Whitepaper-Website-Contract correlation.	Passed
Front Running.	Passed

Owner privileges

META VERSE MARKET BEP20 TOKEN Contract

function will transfer token for a specified address recipient is the address to transfer. “amount” is the amount to be transferred.

```
function transfer(address recipient, uint256 amount) public override returns (bool) {
    _transfer(_msgSender(), recipient, amount);
    return true;
}
```

Transfers ownership of the contract to a new account (`newOwner`). Can only be called by the current owner.

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
}
```

Transfer tokens from the “from” account to the “to” account. The calling account must already have sufficient tokens approved for spending from the “from” account and “From” account must have sufficient balance to transfer.” Spender” must have sufficient allowance to transfer.

```
function transferFrom(address sender, address recipient, uint256 amount) public override returns (bool) {
    _transfer(sender, recipient, amount);
    _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount,
     "ERC20: transfer amount exceeds allowance"));
    return true;
}
```

Approve the passed address to spend the specified number of tokens on behalf of msg. sender. “spender” is the address which will spend the funds. “tokens” the number of tokens to be spent. Beware that changing an allowance with this method brings the risk that someone may use both the old and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards.
https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md recommends that there are no checks for the approval double-spend attack as this should be implemented in user interfaces.

```
function approve(address spender, uint256 amount) public override returns (bool) {
    _approve(_msgSender(), spender, amount);
    return true;
}
```

This will decrease approval number of tokens to spender address. "_spender" is the address whose allowance will decrease and "_subtractedValue" are number of tokens which are going to be subtracted from current allowance.

```
function decreaseAllowance(address spender, uint256 subtractedValue) public virtual
returns (bool) {
    _approve(_msgSender(), spender, _allowances[_msgSender()][spender].
    sub(subtractedValue, "ERC20: decreased allowance below zero"));
    return true;
}
```

Owner of the contract exclude account from reward. Pancake router cannot excluded from reward.

```
function excludeFromReward(address account) public onlyOwner() {

    require(!_isExcluded[account], "Account is already excluded");
    if(_rOwned[account] > 0) {
        _tOwned[account] = tokenFromReflection(_rOwned[account]);
    }
    _isExcluded[account] = true;
    _excluded.push(account);
}
```

Owner of the contract exclude account from fees.

```
function excludeFromFee(address account) public onlyOwner {
    _isExcludedFromFee[account] = true;
}
```

This will increase approval number of tokens to spender address. "_spender" is the address whose allowance will increase and "_addedValue" are number of tokens which are going to be added in current allowance. approve should be called when allowed[_spender] == 0. To increment allowed value is better to use this function to avoid 2 calls (and wait until the first transaction is mined).

```
function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
    _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
    return true;
}
```

Leaves the contract without owner. It will not be possible to call `onlyOwner` functions anymore. Can only be called by the current owner. Renouncing ownership will leave the contract without an owner, thereby removing any functionality that is only available to the owner.

```
function renounceOwnership() public virtual onlyOwner {
    emit OwnershipTransferred(_owner, address(0));
    _owner = address(0);
}
```

Owner of this contract add the addresses in reward.

```
function includeInReward(address account) external onlyOwner() {
    require(_isExcluded[account], "Account is already excluded");
    for (uint256 i = 0; i < _excluded.length; i++) {
        if (_excluded[i] == account) {
            _excluded[i] = _excluded[_excluded.length - 1];
            _tOwned[account] = 0;
            _isExcluded[account] = false;
            _excluded.pop();
            break;
        }
    }
}
```

Owner of this contract include the addresses into fee.

```
function includeInFee(address account) public onlyOwner {
    _isExcludedFromFee[account] = false;
}
```

Locks the contract for owner for the amount of time provided. Owner of this contract is the only executer of this function.

```
function lock(uint256 time) public virtual onlyOwner {
    _previousOwner = _owner;
    _owner = address(0);
    _lockTime = block.timestamp + time;
    emit OwnershipTransferred(_owner, address(0));
}
```

Owner of this contract set the maximum transaction percentage.

```
function setMaxTxAmount(uint256 maxTxAmount) external onlyOwner() {
    _maxTxAmount = maxTxAmount;
}
```

Owner of this contract flip the swap and liquify state.

```
function setSwapAndLiquifyEnabled(bool _enabled) public onlyOwner {
    swapAndLiquifyEnabled = _enabled;
    emit SwapAndLiquifyEnabledUpdated(_enabled);
}
```

Unlocks the contract for owner when _lockTime is exceeds. Only previous owner can call this function and get back ownership and unlock the contract.

```
function unlock() public virtual {
    require(_previousOwner == msg.sender, "You don't have permission to unlock");
    require(block.timestamp > _lockTime , "Contract is locked until 7 days");
    emit OwnershipTransferred(_owner, _previousOwner);
    _owner = _previousOwner;
}
```

Owner of this contract set the contract to presale state. Tax fee, Liquidity fee, previous tax fee, marketing fee, previous marketing fee, developer fee, previous developer fee, burn fee, previous burn fee to "0" and swap and liquify to "false". Maximum transaction amount set to 9000000000.

```
function prepareForPreSale() external onlyOwner
{
    _taxFee = 0;
    _previousTaxFee = 0;
    _liquidityFee = 0;
    _previousLiquidityFee = 0;
    _marketingFee = 0;
    _previousMarketingFee = 0;
    _developerFee = 0;
    _previousDeveloperFee = 0;
    _burnFee = 0;
    _previousBurnFee = 0;
    setSwapAndLiquifyEnabled(false);
    _maxTxAmount = 9_000_000_000 * 10**9;
}
```

Owner of this contract set the burn fee percentage.

```
function setBurnFeePercent(uint256 burnFee) external onlyOwner() {
    _burnFee = burnFee;
    _previousBurnFee = burnFee;
}
```

Owner of this contract set the sale tax fee, sale liquidity fee, sale marketing fee, sale developer fee and sale burn fee percentage.

```
function setAllSaleFeesPercents(uint256 taxFee, uint256 liquidityFee,
uint256 marketingFee, uint256 developerFee, uint256 burnFee) external onlyOwner
{
    _saleTaxFee = taxFee;
    _saleLiquidityFee = liquidityFee;
    _saleMarketingFee = marketingFee;
    _saleDeveloperFee = developerFee;
    _saleBurnFee = burnFee;
}
```

Owner of this contract set the fee percentage for developer address to receive.

```
function setDeveloperFeePercent(uint256 developerFee) external onlyOwner() {
    _developerFee = developerFee;
    _previousDeveloperFee = developerFee;
}
```

Owner of this contract set the liquidity fee percentage.

```
function setLiquidityFeePercent(uint256 liquidityFee) external onlyOwner() {
    _liquidityFee = liquidityFee;
    _previousLiquidityFee = liquidityFee;
}
```

Owner of this contract set the marketing fee percentage.

```
function setMarketingFeePercent(uint256 marketingFee) external onlyOwner() {
    _marketingFee = marketingFee;
    _previousMarketingFee = marketingFee;
}
```

Owner of this contract set the marketing fee receiver address.

```
function setMarketingAddress(address _marketingAddress) external onlyOwner() {
    marketingAddress = payable(_marketingAddress);
}
```

Owner of this contract set the tax fee percentage.

```
function setTaxFeePercent(uint256 taxFee) external onlyOwner() {
    _taxFee = taxFee;
    _previousTaxFee = taxFee;
}
```

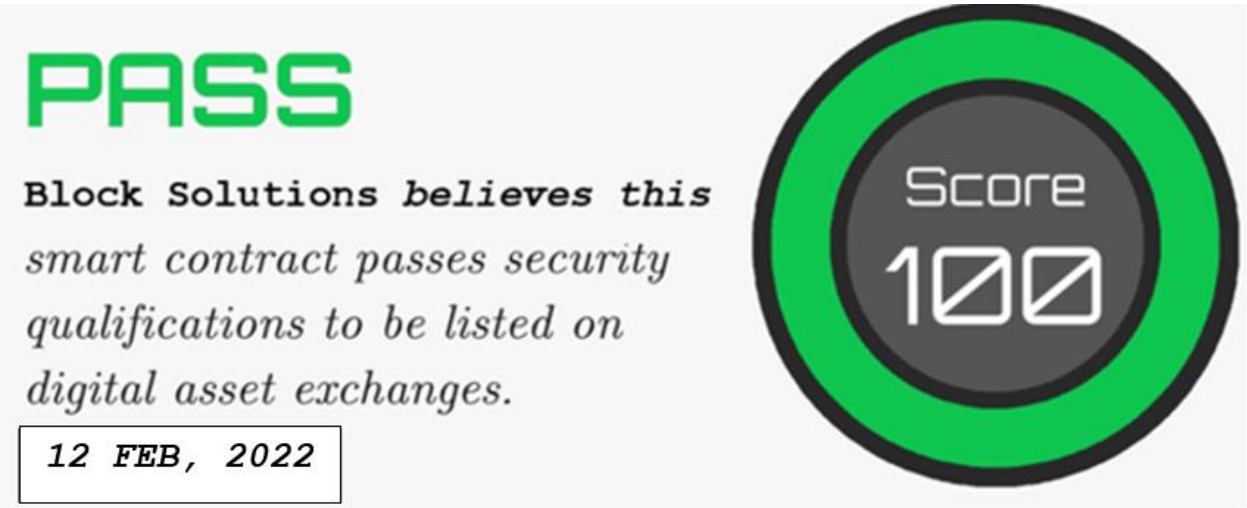
Owner of this contract call this function to start public sale by applying all the feeses.

```
function afterPreSale() external onlyOwner
{
    _taxFee = 3;
    _previousTaxFee = _taxFee;

    _liquidityFee = 3;
    _previousLiquidityFee = _liquidityFee;

    _marketingFee = 2;
    _previousMarketingFee = _marketingFee;

    _developerFee = 1;
    _previousDeveloperFee = _developerFee;

    _burnFee = 0;
    _previousBurnFee = _burnFee;

    setSwapAndLiquifyEnabled(true);
    _maxTxAmount = 9_000_000_0 * 10**9;
}
```

Allows a 'standard'/non-excluded from reflection address to reduce its reflected token amount in order to increase all the others non-excluded addresses balances (via the elastic supply ratio. The second to last line in deliver _rTotal = _rTotal.sub(rAmount); means the "total amount of reflection" decreases by the equivalent of amount in reflected token. This increases the ratio 'token/reflected token' that is used to computed any user balance

```
function deliver(uint256 tAmount) public {
    address sender = _msgSender();
    require(!_isExcluded[sender], "Excluded addresses cannot call this function");
    (uint256 rAmount,,,,,) = _getValues(tAmount);
    _rOwned[sender] = _rOwned[sender].sub(rAmount);
    _rTotal = _rTotal.sub(rAmount);
    _tFeeTotal = _tFeeTotal.add(tAmount);
}
```

Testing Summary



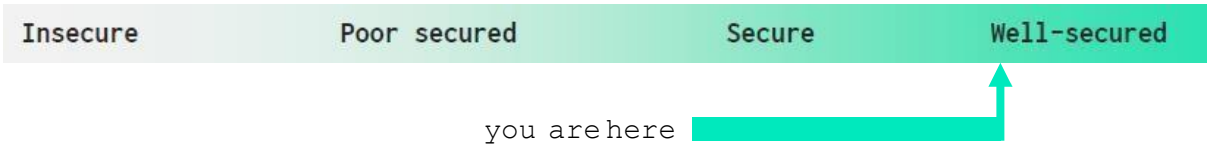
Quick Stats:

Main Category	Subcategory	Result
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
	Var. 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
Business Risk	The maximum limit for mintage not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: PASSED

Executive Summary

According to the standard audit assessment, Customer`s solidity smart contract is Well-secured.
Again, it is recommended to perform an Extensive audit assessment to bring a more assured conclusion.



We used various tools like Mythril, Slither and Remix IDE. At the same time this finding is based on critical analysis of the manual audit.
All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Quick Stat section.
We found 0 critical, 0 high, 0 medium and 2 low level issues.

Code Quality

The META VERSE MARKET BEP20 TOKEN Smart Contract protocol consists of one smart contract. It has other inherited contracts like Context, IERC20, Ownable. These are compact and well written contracts. Libraries used in META VERSE MARKET BEP20 TOKEN Smart Contract are part of its logical algorithm. They are smart contracts which contain reusable code. Once deployed on the blockchain (only once), it is assigned a specific address and its properties / methods can be reused many times by other contracts in protocol. The BLOCKSOLUTIONS team has **not** provided scenario and unit test scripts, which would help to determine the integrity of the code in an automated way.
Overall, the code is not commented. Commenting can provide rich documentation for functions, return variables and more.

Documentation

As mentioned above, it's recommended to write comments in the smart contract code, so anyone can quickly understand the programming flow as well as complex code logic. We were given a META VERSE MARKET BEP20 TOKEN Smart Contract smart contract code in the form of File.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure that are based on well-known industry standard open-source projects. And even core code blocks are written well and systematically. This smart contract does not interact with other external smart contracts.

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial functions
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical

No critical severity vulnerabilities were found.

High

No high severity vulnerabilities were found.

Medium

No Medium severity vulnerabilities were found.

Low

(1) Approve ()

Approve the passed address to spend the specified number of tokens on behalf of msg. sender. “spender” is the address which will spend the funds. “amount” the number of tokens to be spent. Beware that changing an allowance with this method brings the risk that someone may use both the old and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards.
https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md recommends that there are no checks for the approval double-spend attack as this should be implemented in user interfaces.

```
function approve(address spender, uint256 amount) public override returns (bool) {
    _approve(_msgSender(), spender, amount);
    return true;
}
```

(2) IncreaseAllowance ()

This will increase approval number of tokens to spender address. “spender” is the address whose allowance will increase and “addedValue” are number of tokens which are going to be added in current allowance. approve should be called when _allowances[spender] == 0. To increment allowed value is better to use this function to avoid 2 calls (and wait until the first transaction is mined.

```
function increaseAllowance(address spender, uint256 addedValue) public
virtual returns (bool) {
    _approve(_msgSender(), spender, _allowances[_msgSender()][spender].
    add(addedValue));
    return true;
}
```

Solution: This issue is acknowledged.

Conclusion

The Smart Contract code passed the audit successfully with some considerations to take. There were two low severity warnings raised meaning that they should be taken into consideration but if the confidence in the owner is good, they can be dismissed. The last change is advisable in order to provide more security to new holders. Nonetheless this is not necessary if the holders and/or investors feel confident with the contract owners. We were given a contract code. And we have used all possible tests based on given objects as files. So, it is good to go for production.
Since possible test cases can be unlimited for such extensive smart contract protocol, hence we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything. Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high-level description of functionality was presented in Quick Stat section of the report.
Audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security state of the reviewed contract is "Well Secured".

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are 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.

Manual Code Review:

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included 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, skim open issue tickets, and generally 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

generally, 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.

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.