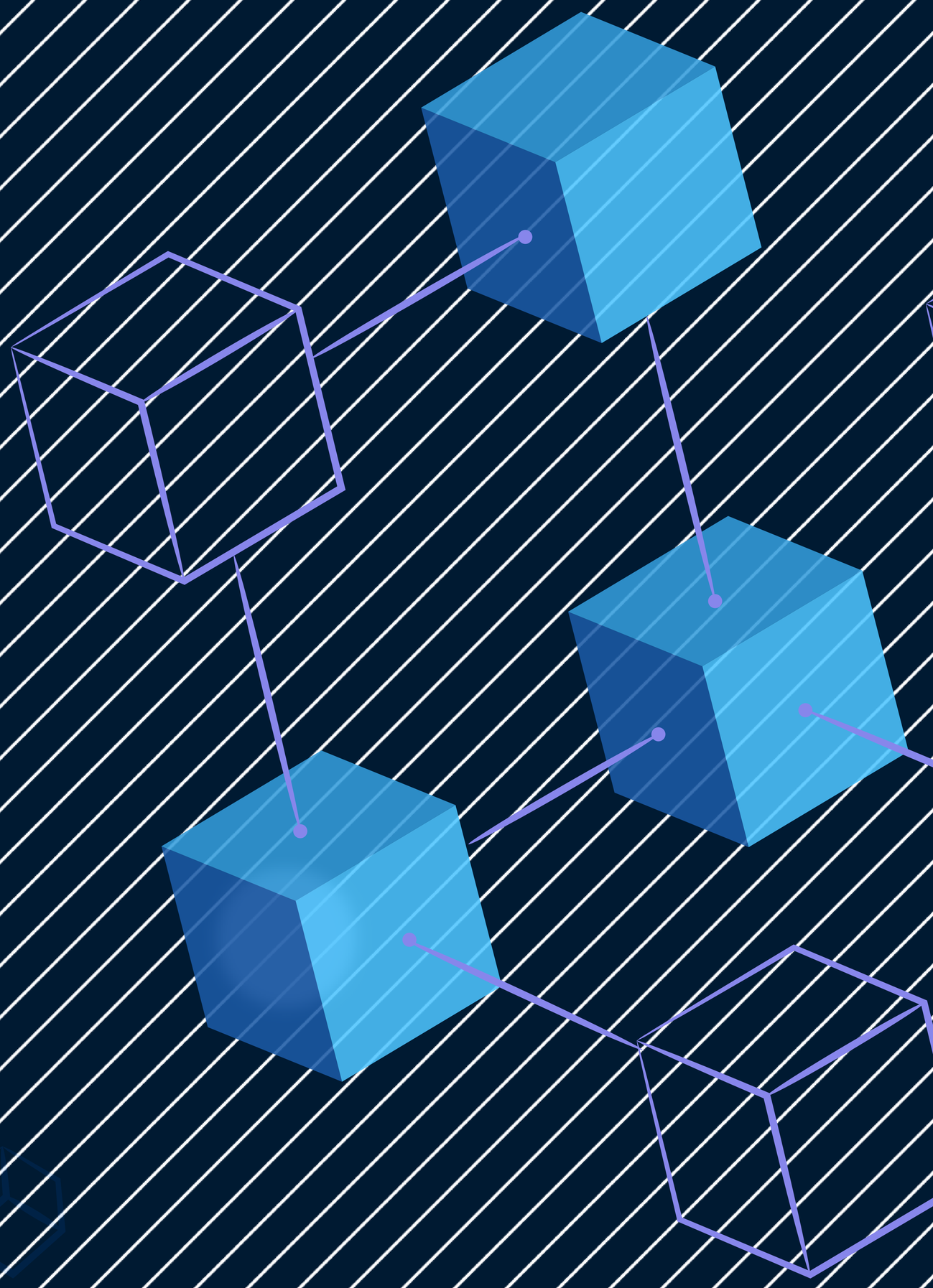
QuillAudits

Audit Report
October, 2021



For

UniFarm

by OroPocket

Contents

Contents

Contents

Scope of the Audit

The scope of this audit was to analyze and document the Unifarm smart contracts 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
Mythril, Slither, SmartCheck, Surya, Solhint.

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 four 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	1	5	0	2
Closed	0	2	10	0

Introduction

During the period of **September 25, 2021, to October 06, 2021** - QuillAudits Team performed a security audit for **Unifarm** smart contracts.

The code for the audit was taken from the following official repo of **Unifarm**:
https://github.com/themohitmadan/unifarm_amm

Note	Date	Commit hash
Version 1	September	02d334ee5c22107d1a4bf548180afb5873e50a1e
Version 2	October	c82cf85d0fabd25dc25fd0eca3dca0325199d64c

Issues Found

A. Contract – UniswapV2ERC20

High severity issues

No issues were found.

Medium severity issues

A.1 Approve Race

```
Line 81:
function approve(address spender, uint256 value) external returns (bool) {
        _approve(msg.sender, spender, value);
        return true;
    }
```

Description

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 a 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 will be able to get both approval amounts of both transactions.

Remediation

Use increaseAllowance and decreaseAllowance functions that are available in ERC20 in order to modify the allowance instead of calling the approve function to override it.

Solved

The Unifarm team has solved the issue in version 2 by adding the increaseAllowance and the decreaseAllowance functions to the smart contract code.

A.2 Missing Address Verification

```
Line 81:
function approve(address spender, uint256 value) external returns (bool) {
        _approve(msg.sender, spender, value);
        return true;
    }
```

```
Line 86:
function transfer(address to, uint256 value) external returns (bool) {
        _transfer(msg.sender, to, value);
        return true;
    }
```

```
Line 91:
function transferFrom(from, to, uint256 value) external returns (bool) {
        if (allowance[from][msg.sender] != uint256(-1)) {allowance[from][msg.sender] =
        allowance[from][msg.sender].sub(value);
        }
        _transfer(from, to, value);
        return true;
    }
```

```
Line 103:
    function permit(address owner, address spender, uint256 value,
uint256 deadline, uint8 v, bytes32 r, bytes32 s) external {
require(deadline >= block.timestamp, 'UniswapV2: EXPIRED');
bytes32 digest = keccak256(
    abi.encodePacked(\x19\x01', DOMAIN_SEPARATOR,
        keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value,
          nonces[owner]++, deadline)))
);
```

Description

Certain functions lack a safety check in the address, the address-type argument should include a zero-address test, otherwise, the contract's functionality may become inaccessible or tokens may be burned in perpetuity.

Remediation

It’s recommended to undertake further validation prior to user-supplied data. The concerns can be resolved by utilizing a whitelist technique or a modifier.

Solved

The Unifarm team has solved the issue in version 2 by adding require statements to verify the addresses that are passed in the arguments.

A.3 Usage of block.timestamp

```
Line 112:
require(deadline >= block.timestamp, 'UniswapV2: EXPIRED');
```

Description
Block.timestamp is used in the contract. The variable block is a set of variables. The timestamp does not always reflect the current time and may be inaccurate. The value of a block can be influenced by miners. Maximal Extractable Value attacks require a timestamp of up to 900 seconds. There is no guarantee that the value is right, all that is guaranteed is that it is higher than the timestamp of the previous block.

Remediation
You can use an Oracle to get the exact time or verify if a delay of 900 seconds won't impact the logic of the smart contract.

Acknowledged
The Unifarm team has acknowledged the risk knowing that 900 seconds delay won't affect the business logic.

Low severity issues

A.4 Integer overflow

```
Line 117:
keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value,
  nonces[owner]++, deadline))
```

Description
The nonce variable in an uint256, incrementing this value in the solidity version that is < 0.8.0 without any verification can result in an overflow so if the value reached 2^256-1 incrementing it will change the nonce value to 0

Remediation
Use the add function from the SafeMath library. Also returning an error message would help explain why the transaction failed.

Solved
The Unifarm team has solved the issue in version 2 by using the SafeMath library to perform the addition operation.

Informational Issues

A.5 Use of Inline Assembly

```
Line 27:
        assembly {
            chainId := chainid
        }
```

Description
Inline assembly is a way to access the EVM at a low level. This discards several important safety features in Solidity.

Remediation
When possible, do not use inline assembly because it is a way to access the EVM at a low level. An attacker could bypass many important safety features of Solidity.

Acknowledged
The Unifarm team has acknowledged the information knowing that it doesn’t represent a specific risk to the smart contract.

B. Contract – UniswapPair

High severity issues

No issues were found.

Medium severity issues

B.1 Usage of block.timestamp

```
Line 96:
uint32 blockTimestamp = uint32(block.timestamp % 2**32);
```

Description

Block.timestamp is used in the contract. The variable block is a set of variables. The timestamp does not always reflect the current time and may be inaccurate. The value of a block can be influenced by miners. Maximal Extractable Value attacks require a timestamp of up to 900 seconds. There is no guarantee that the value is right, all that is guaranteed is that it is higher than the timestamp of the previous block.

Remediation

You can use an Oracle to get the exact time or verify if a delay of 900 seconds won't impact the logic of the smart contract.

Acknowledged

The Unifarm team has acknowledged the risk knowing that 900 seconds delay won't affect the business logic.

Low severity issues

B.2 Missing Address Verification

```
Line 77:
function initialize(address _token0, address _token1,
address _trustedForwarder ) external {
        require(msg.sender == factory, 'UniswapV2: FORBIDDEN');
        token0 = _token0;
        token1 = _token1;

    trustedForwarder = _trustedForwarder;
    }
```

```
Line 259:
function skim(address to) external lock {
        address _token0 = token0;
        address _token1 = token1;
        _safeTransfer(_token0, to, IERC20(_token0).balanceOf(address(this))
      .sub(reserve0));
        _safeTransfer(_token1, to, IERC20(_token1).balanceOf(address(this))
      .sub(reserve1));
    }
```

```
Line 143:
function mint(address to) external lock returns (uint liquidity) {
        (uint112 _reserve0, uint112 _reserve1,) = getReserves();
        uint balance0 = IERC20(token0).balanceOf(address(this));
        uint balance1 = IERC20(token1).balanceOf(address(this));
```

Description

Certain functions lack a safety check in the address, the address-type argument should include a zero-address test, otherwise, the contract's functionality may become inaccessible or tokens may be burned in perpetuity.

Remediation

It's recommended to undertake further validation prior to user-supplied data. The concerns can be resolved by utilizing a whitelist technique or a modifier.

Solved

The Unifarm team has solved the issue in version 2 by adding require statements to verify the addresses that are passed in the arguments.

C. Contract – UnifarmFactory

High severity issues

No issues were found.

Medium severity issues

C.1 Race Condition

```
Line 84:
function updateLPFeeConfig(address _pairAddress, bool _feeInToken,
        uint256 _fee) external onlyOwner {
            Pair storage pair = pairConfigs[_pairAddress];
            pair.lpFeesInToken = _feeInToken;
            pair.lpFee = _fee;
        }
        function updateSwapFeeConfig(address _pairAddress, bool _feeInToken,
        uint256 _fee) external onlyOwner {
            //To set max and min limit for both fee types
            Pair storage pair = pairConfigs[_pairAddress];
            pair.lpFeesInToken = _feeInToken;
            pair.lpFee = _fee;
        }
```

Description
If the owner updates the fee there is a possibility that his transaction will be mined before a user transaction that mints the fee, that will make him use a fee that is different than the one stored in the smart contract.

Remediation
Add the fee as the argument to both functions and add a require that verifies that the fee provided in the arguments is the same as the one that is stored in the smart contract.

Acknowledged
The Unifarm team has acknowledged the risk.

Low severity issues

C.2 Missing Address Verification

```
Line 24:
constructor(address payable _feeTo, address _trustedForwarder,
        uint256 _lpFee, uint256 _swapFee, bool _lpFeesInToken,
        bool _swapFeesInToken) public {
            require(_feeTo != address(0), 'Unifarm: WALLET_ZERO_ADDRESS');
            feeTo = _feeTo;
            Pair memory pair;
            pair.lpFeesInToken = _lpFeesInToken;
            pair.swapFeesInToken = _swapFeesInToken;
            pair.lpFee = _lpFee;
            pair.swapFee = _swapFee;
            trustedForwarder = _trustedForwarder;
         pairConfigs[address(0)] = pair;
        }
```

```
Line 80:
function setFeeTo(address payable _feeTo) external onlyOwner {
            feeTo = _feeTo;
        }
```

Description

Certain functions lack a safety check in the address, the address-type argument should include a zero-address test, otherwise, the contract's functionality may become inaccessible or tokens may be burned in perpetuity.

Remediation

It’s recommended to undertake further validation prior to user-supplied data. The concerns can be resolved by utilizing a whitelist technique or a modifier.

Solved

The Unifarm team has solved the issue in version 2 by adding require statements to verify the addresses that are passed in the arguments.

C.3 Missing Value Verification

```
Line 84:
function updateLPFeeConfig(address _pairAddress, bool _feeInToken,
        uint256 _fee) external onlyOwner {
        //To set max and min limit for both fee types
        Pair storage pair = pairConfigs[_pairAddress];
        pair.lpFeesInToken = _feeInToken;
        pair.lpFee = _fee;
    }

    function updateSwapFeeConfig(address _pairAddress, _feeInToken,
        uint256 _fee) external onlyOwner {
        //To set max and min limit for both fee types
        Pair storage pair = pairConfigs[_pairAddress];
        pair.lpFeesInToken = _feeInToken;
        pair.lpFee = _fee;
    }
```

Description

Certain functions lack a safety check in the argument's value, the _fee argument should be lower than 1000.

Remediation

Add a require or modifier in order to verify that the input value is lower than 10000.

Solved

The Unifarm team has solved the issue in version 2 by adding require statements to verify the values that are passed in the arguments.

C.4 Renounce Ownership

```
Line 8:
contract UnifarmFactory is IUnifarmFactory, Ownable, BaseRelayRecipien
```

Description

Typically, the contract's owner is the account that deploys the contract. As a result, the owner is able to perform certain privileged activities on his behalf. The renounceOwnership function is used in smart contracts to renounce ownership. Otherwise, if the contract's ownership has not been transferred previously, it will never have an Owner, which is risky.

Remediation
It is advised that the Owner cannot call renounceOwnership without first transferring ownership to a different address. Additionally, if a multi-signature wallet is utilized, executing the renounceOwnership method for two or more users should be confirmed. Alternatively, the Renounce Ownership functionality can be disabled by overriding it.

Acknowledged
The Unifarm team has acknowledged the issue knowing that they will use the MultiSig wallet contract to prevent from the risk.

Informational Issues

C.5 Use of Inline Assembly

```
Line 66:
        assembly {
            pair := create2(0, add(bytecode, 32), mload(bytecode), salt)
        }
```

Description
Inline assembly is a way to access the EVM at a low level. This discards several important safety features in Solidity.

Remediation
When possible, do not use inline assembly because it is a way to access the EVM at a low level. An attacker could bypass many important safety features of Solidity.

Acknowledged
The Unifarm team has acknowledged the information knowing that it doesn’t represent a specific risk to the smart contract.

D. Contract – GovernorBravoDelegate

High severity issues

No issues were found.

Medium severity issues

D.1 Usage of block.timestamp

```
Line 187:
Proposal storage proposal = proposals[proposalId];
      uint256 eta = add256(block.timestamp, timelock.delay());
```

```
Line 319:
}else if(block.timestamp >= add256(proposal.eta,timelock.GRACE_PERIOD())){
            return ProposalState.Expired;
```

Description

Block.timestamp is used in the contract. The variable block is a set of variables. The timestamp does not always reflect the current time and may be inaccurate. The value of a block can be influenced by miners. Maximal Extractable Value attacks require a timestamp of up to 900 seconds. There is no guarantee that the value is right, all that is guaranteed is that it is higher than the timestamp of the previous block.

Remediation

You can use an Oracle to get the exact time or verify if a delay of 900 seconds won't impact the logic of the smart contract.

Acknowledged

The Unifarm team has acknowledged the risk knowing that 900 seconds delay won't affect the business logic.

Low severity issues

D.2 Missing Address Verification

```
Line 457:
function initialize(address timelock_,address ufarm_,
        uint256 votingPeriod_,uint256 votingDelay_,uint256 proposalThreshold_,
        address trustedForwarder_) public {
        require(address(timelock) == address(0), 'GovernorBravo::initialize:
      can only initialize once');
        require(_msgSender() == admin, 'GovernorBravo::initialize: admin only');
        require(timelock_ != address(0), 'GovernorBravo::initialize: invalid
    timelock address');
        require(ufarm_ != address(0), 'GovernorBravo::initialize: invalid ufarm          address');
```

Description

Certain functions lack a safety check in the address, the address-type argument trustedForwarder should include a zero-address test, otherwise, the contract's functionality may become inaccessible or tokens may be burned in perpetuity.

Remediation

It’s recommended to undertake further validation prior to user-supplied data. The concerns can be resolved by utilizing a whitelist technique or a modifier.

Solved

The Unifarm team has solved the issue in version 2 by adding require statements to verify the addresses that are passed in the arguments.

D.3 Integer overflow

```
Line 87:
function _initiate() external {
        require(_msgSender() == admin, 'GovernorBravo::_initiate: admin only');
        require(initialProposalId == 0, 'GovernorBravo::_initiate: can only
          initiate once');
        initialProposalId = 1;
        proposalCount++;
        timelock.acceptAdmin();
        }
```

Description
The proposalCount variable in an uint256, incrementing this value in the solidity version that is < 0.8.0 without any verification can result in an overflow so if the value reached 2^256-1 incrementing it will change the nonce value to 0.

Remediation
Use the add function from the SafeMath library. Also returning an error message would help explain why the transaction failed.

Solved
The Unifarm team has solved the issue in version 2 by using the SafeMath library to perform the addition operation.

E. Contract – AMMUtility

High severity issues

E.1 Use of transferFrom instead of safeTransferFrom

```
Line 33:
IERC20(_sourceToken).transferFrom(
    _userAddress, address(this), _amount);
uint256 tokensReceived = _swap(_sourceToken, _destToken, _amount);
IERC20(_destToken).transfer(_userAddress, tokensReceived);
emit TokenSwapExecuted(_sourceToken, _destToken, tokensReceived);
```

Description

The ERC20 standard token implementation functions also return the transaction status as a Boolean. It's good practice to check for the return status of the function call to ensure that the transaction was successful. It is the developer's responsibility to enclose these function calls with require() to ensure that, when the intended ERC20 function call returns false, the caller transaction also fails. However, it is mostly missed by developers when they carry out checks. in effect, the transaction would always succeed, even if the token transfer didn't.

Remediation

Implement SafeERC20 in order to use the safeTransfer function instead of transfer.

Solved

The Unifarm team has solved the issue in version 2 by using the safeTransfer function instead of transfer.

Medium severity issues

No issues were found.

Low severity issues

E.2 Missing Address Verification

```
Line 14:
constructor(address payable _feeTo, uint256 _fee) public {
        fee = _fee;
        feeTo = _feeTo;
        }
```

```
Line 19:
function swapTokens(address _userAddress, address _sourceToken,
        address _destToken, uint256 _amount) external payable {
        require(_sourceToken != address(0) && _destToken != address(0),
      'AMMUtility: Invalid token addresses');
        require(_sourceToken != _destToken, 'AMMUtility:Both address are same');
        require(_amount != 0, 'AMMUtility: Invalid token amount');
        require(msg.value >= fee, 'AMMUtility: Fee not received');
```

Description

Certain functions lack a safety check in the address, the address-type argument _userAddress should include a zero-address test, otherwise, the contract's functionality may become inaccessible or tokens may be burned in perpetuity.

Remediation

It’s recommended to undertake further validation prior to user-supplied data. The concerns can be resolved by utilizing a whitelist technique or a modifier.

Solved

The Unifarm team has solved the issue in version 2 by adding require statements to verify the addresses that are passed in the arguments.

F. Contract – UniswapV2Router02

High severity issues

No issues were found.

Medium severity issues

F.1 Usage of block.timestamp

```
Line 18:
modifier ensure(uint256 deadline) {
        require(deadline >= block.timestamp, 'UnifarmRouter: EXPIRED');
        _;
    }
```

Description

Block.timestamp is used in the contract. The variable block is a set of variables. The timestamp does not always reflect the current time and may be inaccurate. The value of a block can be influenced by miners. Maximal Extractable Value attacks require a timestamp of up to 900 seconds. There is no guarantee that the value is right, all that is guaranteed is that it is higher than the timestamp of the previous block.

Remediation

You can use an Oracle to get the exact time or verify if a delay of 900 seconds won't impact the logic of the smart contract.

Acknowledged

The Unifarm team has acknowledged the risk knowing that 900 seconds delay won't affect the business logic.

F.2 For loop over dynamic array

```
Line 252:
function _swap(uint256[] memory amounts, address[] memory path,
address _to) internal virtual {
        for (uint256 i; i < path.length - 1; i++) {
            (address input, address output) = (path[i], path[i + 1]);
            (address token0, ) = UnifarmLibrary.sortTokens(input, output);
            uint256 amountOut = amounts[i + 1];
            (uint256 amount0Out, uint256 amount1Out) = input == token0
                ? (uint256(0), amountOut): (amountOut, uint256(0));
            address to = i < path.length - 2 ?
      UnifarmLibrary.pairFor(factory, output, path[i + 2]) : _to;
            IUnifarmPair(UnifarmLibrary.pairFor(factory, input, output))
                .swap(amount0Out, amount1Out, to, new bytes(0));
        }
    }
```

```
Line 376:
function _swapSupportingFeeOnTransferTokens(address[] memory path,
      address _to) internal virtual {
            for (uint256 i; i < path.length - 1; i++) {
            // [+] loop over dynamic array
                  (address input, address output) = (path[i], path[i + 1]);
                  (address token0, ) = UnifarmLibrary.sortTokens(
                                    input, output);
                  IUnifarmPair pair = IUnifarmPair(UnifarmLibrary.pairFor(
                                    factory, input, output));
               uint256 amountInput;
               uint256 amountOutput;
               {
            // scope to avoid stack too deep errors
               (uint256 reserve0, uint256 reserve1, ) = pair.getReserves();
            (uint256 reserveInput, uint256 reserveOutput) = input == token0
               ? (reserve0, reserve1): (reserve1, reserve0);
            amountInput = IERC20(input).balanceOf(address(pair))
          .sub(reserveInput);
            amountOutput = UnifarmLibrary.getAmountOut(
                        amountInput, reserveInput, reserveOutput);
      }
      (uint256 amount0Out, uint256 amount1Out) = input == token0
            ? (uint256(0), amountOutput): (amountOutput, uint256(0));
            address to = i < path.length - 2 ?
            UnifarmLibrary.pairFor(factory, output, path[i + 2]) : _to;
            pair.swap(amount0Out, amount1Out, to, new bytes(0));
         }
      }
```

Description
When smart contracts are deployed or their associated functions are invoked, the execution of these operations always consumes a certain quantity of gas, according to the amount of computation required to accomplish them. Modifying an unknown-size array that grows in size over time can result in a Denial-of-Service attack. Simply by having an excessively huge array, users can exceed the gas limit, therefore preventing the transaction from ever succeeding.

Remediation
Avoid actions that involve looping across the entire data structure. If you really must loop over an array of unknown size, arrange for it to consume many blocs and thus multiple transactions.

Solved
The Unifarm team has solved the issue in version 2 by limiting the array's size from the front-end.

Low severity issues

F.3 Missing Address Verification

```
Line 23:
        constructor(address _factory, address _WETH) public {
            factory = _factory;
            WETH = _WETH;
        }
```

Description
Certain functions lack a safety check in the address, the address-type arguments should include a zero-address test, otherwise, the contract's functionality may become inaccessible or tokens may be burned in perpetuity.

Remediation
It's recommended to undertake further validation prior to user-supplied data. The concerns can be resolved by utilizing a whitelist technique or a modifier.

Solved
The Unifarm team has solved the issue in version 2 by adding require statements to verify the addresses that are passed in the arguments.

G. Contract – MultiSigWallet

High severity issues

No issues were found.

Medium severity issues

No issues were found.

Low severity issues

No issues were found.

G.1 Integer overflow

```
Line 336:
function addTransaction(address destination, uint256 value,
bytes memory data) internal notNull(destination)
returns (uint256 transactionId) {
transactionId = transactionCount;
transactions[transactionId] = Transaction({
    destination: destination,
    value: value,
    data: data,
    executed: false
});
transactionCount += 1;
emit Submission(transactionId);
}
```

```
Line 361:
function getConfirmationCount(uint256 transactionId) public view
returns (uint256 count) {
        for (uint256 i = 0; i < owners.length; i++)
        if (confirmations[transactionId][owners[i]]) count += 1;
    }
```

Description
The transactionCount variable in an uint256, incrementing this value in the solidity version that is < 0.8.0 without any verification can result in an overflow so if the value reached 2^256-1 incrementing it will change the nonce value to 0.

Remediation
Use the add function from the SafeMath library. Also returning an error message would help explain why the transaction failed.

Solved
The Unifarm team has solved the issue in version 2 by using the SafeMath library to perform the addition operation.

Automated Tests

The automated tests can be found here - https://docs.google.com/document/d/16eCQuXL0W9l_3Sm3BSdWjR5WPabrAc5eWp3EXC7VsF0/edit?usp=sharing

Results

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

Unit Test

We execute unit tests for the different contracts. Below is the report:

```
root@Blockchain:~/unifarm_amm-7a24890be9d9eea6eb9ee5365c0d306123811563# npx hardhat test
Downloading compiler 0.5.16
Compiling 13 files with 0.5.16
contracts/governance/GovernorBravoInterfaces.sol:2:1: Warning: Experimental features are turned on. Do not use experimental features on live deployments.
pragma experimental ABIEncoderV2;
^-------------------------------^

contracts/governance/GovernorBravoDelegate.sol:2:1: Warning: Experimental features are turned on. Do not use experimental features on live deployments.
pragma experimental ABIEncoderV2;
^-------------------------------^

Downloading compiler 0.6.6
Compiling 4 files with 0.6.6
contracts/utility/AMMUtility.sol:42:9: Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
        address _sourceToken,
        ^------------------^

contracts/utility/AMMUtility.sol:43:9: Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
        address _destinationToken,
        ^-----------------------^

contracts/utility/AMMUtility.sol:44:9: Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
        uint256 _amount
        ^-------------^

contracts/utility/AMMUtility.sol:41:5: Warning: Function state mutability can be restricted to pure
    function _swap(
    ^ (Relevant source part starts here and spans across multiple lines).

contracts/utility/UniswapV2Router02.sol:12:1: Warning: Contract code size exceeds 24576 bytes (a limit introduced in Spurious Dragon). This contract may not be deployable on mainnet. Consider enabling the opt
zer (with a low "runs" value!), turning off revert strings, or using libraries.
contract UnifarmRouter02 is IUnifarmRouter02 {
^ (Relevant source part starts here and spans across multiple lines).

Downloading compiler 0.8.0
Compiling 9 files with 0.8.0
Compilation finished successfully


  AMMUtility
    ✓ swap (325ms)

  governorBravo#castVote/2
    We must revert if:
      ✓ There does not exist a proposal with matching proposal id where the current block number is between the proposal's start block (exclusive) and end block (inclusive)
      ✓ Such proposal already has an entry in its voters set matching the sender (89ms)
    Otherwise
      ✓ we add the sender to the proposal's voters set (46ms)
      ✓ receipt uses two loads (205ms)
      ✓ we take the balance returned by GetPriorVotes for the given sender and the proposal's start block, which may be zero,
```

```
    and we take the balance returned by GetPriorVotes for the given sender and the proposal's start block, which may be zero,
      ✓ and we add that ForVotes (140ms)
      ✓ or AgainstVotes corresponding to the caller's support flag. (115ms)
    castVoteBySig
      ✓ reverts if the signatory is invalid

MultiSigWallet
  ✓ test execution after requirements changed (137ms)

GovernorBravo#queue/1
  overlapping actions
    ✓ reverts on queueing overlapping actions in same proposal (544ms)

UnifarmERC20
  ✓ name, symbol, decimals, totalSupply, balanceOf, DOMAIN_SEPARATOR, PERMIT_TYPEHASH
  ✓ approve
  ✓ transfer
  ✓ transfer:fail
  ✓ transferFrom
  ✓ transferFrom:max
  ✓ permit

UnifarmFactory
  ✓ feeTo, feeToSetter, allPairsLength
  ✓ createPair (121ms)
  ✓ createPair:reverse (109ms)
  ✓ createPair:gas
  ✓ setFeeTo

UnifarmPair
  ✓ mint (96ms)
  ✓ getInputPrice:0 (135ms)
  ✓ getInputPrice:1 (102ms)
  ✓ getInputPrice:2 (113ms)
  ✓ getInputPrice:3 (105ms)
  ✓ getInputPrice:4 (113ms)
  ✓ getInputPrice:5 (117ms)
  ✓ getInputPrice:6 (124ms)
  ✓ optimistic:0 (97ms)
  ✓ optimistic:1 (116ms)
  ✓ optimistic:2 (129ms)
  ✓ optimistic:3 (137ms)
  ✓ swap:gas (132ms)


35 passing (11s)
```

Closing Summary

Overall, smart contracts are very well written and adhere to guidelines. Many issues were discovered during the initial audit; All of them are fixed by the Unifarm Team.

Disclaimer

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

Audit Report
October, 2021

For

UniFarm
by OroPocket

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com