QuillAudits

Audit Report
April, 2024

For

SHARP Token

Table of Content

Table of Content

Executive Summary

Project Name	SHARP
Overview	The SHARP contract is a token contract that has a minimum and maximum transfer limit. The contract owner can add and remove key addresses to the list of admins. These admin addresses have some privileged functions and one of which is updating the maximum or minimum transfer limit of any addresses aside their own address.
Timeline	15th February 2024 - 16th February 2024
Updated Code Received	3rd April 2024
Second Review	5th April 2024 - 10th April 2024
Method	Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
Audit Scope	The scope of this audit was to analyze the MainTokenContract of SHARP Token for quality, security, and correctness.
Source Code	*https://polygonscan.com/address/0xd748F23CBdf67B2deBFa79ed3AD9Ee9e53cbb4b5*
Fixed In	*https://mumbai.polygonscan.com/address/0x312f6dc43776cAd7871ee5222a5143108548A730#code*

Number of Security Issues per Severity

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	2	0	1	8

Checked Vulnerabilities

- 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 contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior 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 analyzed 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

A 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 analyzed, 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 behavior 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

Hardhat, Foundry.

Types of Severity

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.

High Severity Issues

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 Severity Issues

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 Severity Issues

Low-level severity issues can cause minor impact and 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.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

High Severity Issues

1. batchTxnforToken function will revert due to the double present of nonReentrant modifier

Path

MainTokenContract.sol

Function

BatchTxnforToken

```
function BatchTxnforToken(
    address[] calldata recipients,
    uint256 amount
) external nonReentrant returns (bool) {
    require(
        balanceOf(_msgSender()) >= amount * (recipients.length),
        "Insufficient Token Balance"
    );
    for (uint256 i = 0; i < recipients.length; i++) {
        require(
            recipients[i] != address(0),
            "Recipient address cannot be 0"
        );
        _transferToken(recipients[i], amount);
    }
    return true;
}
```

Description

The batchTxnForToken functions are used to perform the batch transfer of tokens to different recipients for either the same or different amounts. However, when these functions are invoked, they will fail because the nonReentrant modifier is attached to the batchTxnForTokens function itself and also in the transfer function which was called in the _transferToken function. The contract execution flow when one of these functions are invoked will work as follows:

1. When the function is called, the first modifier in the function itself will check

```
modifier nonReentrant() {
    // On the first call to nonReentrant, _notEntered will be true
    require(_status != _ENTERED, "ReentrancyGuard: reentrant call");

    // Any calls to nonReentrant after this point will fail
    _status = _ENTERED;

    _;

    // By storing the original value once again, a refund is triggered (see
    // https://eips.ethereum.org/EIPS/eip-2200)
    _status = _NOT_ENTERED;
}
```

2. set the _status to _ENTERED

3. proceed with the check and then the loop in the BatchTxnforTokens

```
require(
    balanceOf(_msgSender()) >= amount * (recipients.length),
    "Insufficient Token Balance"
);
for (uint256 i = 0; i < recipients.length; i++) {
    require(
        recipients[i] != address(0),
        "Recipient address cannot be 0"
    );
    _transferToken(recipients[i], amount);
}
```

4. Within the first loop, the transfer function is called in the _transferToken function but this will revert because of the check in the nonReentrant modifier also added to transfer, the check mentioned in step 1. By this time, status is still _ENTERED.

```
require(_status != _ENTERED, "ReentrancyGuard: reentrant call");
```

Proof of Concept

```
function testBatchTransferTokens() external {
    mintToken();
    // different addresses and same amount
    address[] memory recipients = new address[](4);
    recipients[0] = allowedOne;
    recipients[1] = allowedTwo;
    recipients[2] = userOne;
    recipients[3] = userTwo;

    vm.startPrank(owner);
    vm.expectRevert("ReentrancyGuard: reentrant call");
    sharp.BatchTxnforToken(recipients, 10 ether);
    vm.stopPrank();
}
```

Recommendation

The _transferToken function should be implemented to call the _transfer internal function since this does not have a nonReentrant modifier. This way only the one modifier in the batchTxnforToken function will safeguard the function call.

Status

Resolved

2. Tokens in contract are not redeemable with the withdrawToken function due to wrong use of transfer function

Path

MainTokenContract.sol

Function

withdrawToken

```
function WithdrawToken(      _WithdrawToken  (you)
    uint256 tokenValue
) public onlyOwner nonReentrant returns (bool) {
    require(
        tokenValue > 0 && tokenValue <= _balances[address(this)],
        "Token Requested either exceeds the limit or is 0"
    );
    return _WithdrawToken(_msgSender(), tokenValue);
}
```

Description

The withdrawToken function has two critical issues. First, it shares the same issue as explained in H1 above. The second issue is the lock of funds in the contract due to improper use of the transfer function.

The transfer function called in the _withdrawToken private function takes in a receiver address and the amount to send.

```
function _WithdrawToken(address to, uint256 amount) private returns (bool) {
    bool success = transfer(to, amount);
    if (!success) {
        revert("Token transfer failed");
    }
    emit TokenWithdrawn(to, amount);
    return true;
}
```

In the context of the withdrawToken public function, the onlyOwner address will be the receiver address and also will be the from address instead of the contract address. On inspecting the transfer function, the caller will be the sender and to address that are passed into the _transfer internal function.

```
function transfer(
    address to,
    uint256 amount
)
    public
    virtual
    isAllowedTransfer(amount, msg.sender)
    isAllowedMinimumTransfer(amount, msg.sender)
    nonReentrant
    returns (bool)
{
    address sender = _msgSender();
    _transfer(sender, to, amount);
    if (to != owner() && transferLimit[to] == 0) {
        transferLimit[to] = maxDefaultLimit;
        minTransferLimit[to] = minDefaultLimit;
    }
    return true;
}
```

This implies that the caller is sending the tokens into its own address and tokens in the contract will not be possible to redeem due to this flaw.

Proof of Concept

```
function testWithdrawTokens() external {
    mintToken();

      vm.startPrank(owner);
      // transfer sharp tokens into sharp contract
    sharp.transfer(address(sharp), 20 ether);
    assertEq(sharp.balanceOf(address(sharp)), 20 ether) ;

    // call withdrawToken function
    vm.expectRevert("ReentrancyGuard: reentrant call");
    sharp.WithdrawToken(10 ether);
    vm.stopPrank();
}
```

With two nonReentrant modifier

```
function testWithdrawTokens() external {
    mintToken();

      vm.startPrank(owner);
      // transfer sharp tokens into sharp contract
    sharp.transfer(address(sharp), 20 ether);
    assertEq(sharp.balanceOf(address(sharp)), 20 ether) ;

    // call withdrawToken function
    sharp.WithdrawToken(10 ether);
    vm.stopPrank();

    assertEq(sharp.balanceOf(address(sharp)), 20 ether);
}
```

With one nonReentrant modifier: The tokens sent into the SHARP contract were not sent to the onlyOwner even though the withdrawToken function was successful.

Recommendation

The _withdrawToken function should be implemented to call the _transfer internal function which has no modifier. It should also pass the contract address as the sender to ensure that the tokens are sent from the contract to the onlyOwner address.

Status

Resolved

Medium Severity Issues

No issues were found.

Low Severity Issues

1. Unbounded loop may cause function to revert when array size is large

Description

There are some functions in the contract that loops through an array of parameters passed into it, while some loops are based on the size of the allAllowedAddress. Due to the computational power that will be required based on the size of the array, it is possible that when the function is called, the amount of gas paid might reach the gas limit of a block and invariably revert with all the gas paid. No state changes will reflect but users will pay for massive gas consumption.

Recommendation

Ensure to keep the array size small or add a limit to how many elements that forms an array for a single function call.

Status

Resolved

Informational Issues

1. Inconsistent use of _msgSender from the Context.sol

Path

Ownable.sol and MainTokenContract.sol

Function

- transfer
- transferFrom

Modifier

onlyOwner

Description

In the SHARP token contract and the onlyOwner modifier, msg.sender was used instead. The purpose for using the _msgSender function from the context util contract is to help handle who the real caller of a function is, especially pertaining to a meta-transaction solution.

Recommendation

Consistently maintain the use of _msgSender function from the context contract across all contracts.

Status

Resolved

Reference

https://forum.openzeppelin.com/t/help-understanding-contract-context/10579

2. Adopt conventional naming style for clarity

Path

Ownable.sol and MainTokenContract.sol

Function

- PendingOwner
- startTransferOwnership/StartTransferOwnership
- MinDefaultTransferLimit
- MaxDefaultTransferLimit
- AcceptOwnership

- BatchTxnforToken
- BatchTxnforMatic
- WithdrawToken
- _WithdrawToken

Description

The camelcase naming convention was not used in the contract and some other techniques were not followed. For instance, there are some main functions that begin with a first-letter capitalized name, some do not use an underscore with an internal function to easily differentiate between function visibilities.

Recommendation

Consistently maintain the use of _msgSender function from the context contract across all contracts.

Status

Resolved

3. Remove unused events

Description

```
event TokenTransfer(uint256 indexed amount, address indexed to);
event MaticTransfer(address indexed to, uint256 amount);
```

Some events where declared in token contract but never emitted

Recommendation

Remove these unused events.

Status

Resolved

4. Emit events for critical state changes

Path

MainTokenContract.sol

Function

updateMaxDefaultLimit
updateMinDefaultLimit

Description

This set of functions perform critical changes to two state variables yet do not have any events that track when the variables are updated.

Recommendation

Emit events for maxDefaultLimit and minDefaultLimit state variables.

Status

Resolved

5. Comparison between boolean values

Path

MainTokenContract.sol

Modifier

onlyAllowedAddresses

Description

```
modifier onlyAllowedAddresses() {
    require(
        allowedAddresses[_msgSender()] == true,
        "Not an allowed address"
    );
    _;
}
```

The issue with this modifier is comparing between boolean values. Since the allowedAddresses[_msgSender()] returns a boolean, it is inappropriate to still compare it with a hardcoded “true” value.

Recommendation

Use just the allowedAddresses[_msgSender()]; this can serve the same purpose of knowing if an address is allowed or not.

Status

Resolved

6. Merge two modifiers into one for code clarity

Path

MainTokenContract.sol

Modifier

isAllowedTransfer
isAllowedMininumTransfer

Description

```
modifier isAllowedTransfer(uint256 amount, address sender) {
    require(
        (amount <= transferLimit[sender] || sender == owner()),
        "Cannot transfer more than your limit in one transaction"
    );
    _;
}
```

```
modifier isAllowedMinimumTransfer(uint256 amount, address sender) {
    require(
        (amount >= minTransferLimit[sender] || sender == owner()),
        "Cannot transfer less than your minimum transfer limit"
    );
    _;
}
```

The two modifiers listed can be merged into one modifier and this will still achieve the same purpose. This is also gas efficient.

Recommendation

```
modifier mergedCheck(uint256 amount, address sender) {
    require(
        ((amount >= minTransferLimit[sender] && amount <= transferLimit[sender]) || sender == owner()),
        "Amount not within transfer limit range"
    );
    _;
}
```

Use just the allowedAddresses[_msgSender()]; this can serve the same purpose of knowing if an address is allowed or not.

Status

Resolved

7. Double use of onlyAllowedAddresses increase gas cost or function call

Path

MainTokenContract.sol

Function

_updateMaxTransferLimit
_updateMinTransferLimit

Description

```
    function _updateMaxTransferLimit(
        address addr,
        uint256 limit
    ) internal onlyAllowedAddresses {
        transferLimit[addr] = limit;
        emit UpdateMaxTransferLimit
(addr, limit);
    }
```

```
    function _updateMaxTransferLimit(
        address addr,
        uint256 limit
    ) internal onlyAllowedAddresses {
        transferLimit[addr] = limit;
        emit UpdateMaxTransferLimit
(addr, limit);
    }
```

The functions listed above have the onlyAllowedAddresses modifiers and are invoked in other external functions with the same modifier. Calling these functions will cost more compared to if there is only one modifier to the function.

Recommendation

Remove the modifier from the internal function so one is maintained.

Status

Resolved

8. Add check to functions to prevent receiving empty arrays as parameters

Description

There are some functions in the contract that accept an array of elements to perform execution. If an empty array is passed unknowingly into the function, the call will be successful and gas paid.

Recommendation

Add a require check to these functions to revert for empty arrays.

Status

Resolved

Functional Tests Cases

- Should confirm the effects of the transfer limit when recipients receive token
- Should verify that only allowed addresses can update the default maximum and minimum limit
- Should check that the transfer limit does not apply to the contract address
- Should revert if transfer amount is not within the transfer limit range
- Should batch transfer Matic to different recipients address and same amount
- Should batch transfer Matic to different recipients address and different amount
- Should batch transfer tokens to different addresses
- Should inspect that tokens are sent when ownership is transferred
- Should allow the contract owner to withdraw tokens from the SHARP contract

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 SHARP Token codebase. We performed our audit according to the procedure described above.

Some issues of High, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in SHARP Token smart contract. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of SHARP Token smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the SHARP Token to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

1000+
Audits Completed

$30B
Secured

1M
Lines of Code Audited

Follow Our Journey

Audit Report
April, 2024

For

SHARP Token

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com