DEFIMOON PROJECT
AUDIT AND
DEVELOPMENT

CONTACTS
HTTPS://DEFIMOON.ORG
AUDIT@DEFIMOON.ORG
TELEGRAM
TWITTER

REPORT SMART CONTRACT AUDIT

SYNDIQATE
PROJECT

JULY ,2022

AUDITOR
ARTHUR MACHNACH,
TEAM "DEFIMOON"

APPROVED BY
CYRILL MINYAEV, TEAM
"DEFIMOON"

DEFIMOON

be secure

AUDIT OF PROJECT

HTTPS://SYNDIQATE.IO

HTTPS://TWITTER.COM/SYNDIQATEINFO

HTTPS://T.ME/SYNDIQATE_CHAT

BSC:
0XD286480B95B0A48805B5F3FEF99883ED828F2EF7

Syndiqate project

Disclaimer

This is an internal audit conducted by the Defimoon as a part of its quality control of development, this contract was developed by Defimoon. This audit is not financial, investment, or any other kind of advice and could be used for informational purposes only. This report is not a substitute for doing your own research and due diligence should always be paid in full to any project. Defimoon is not responsible or liable for any loss, damage, or otherwise caused by reliance on this report for any purpose. Defimoon has based this audit report solely on the information provided by the audited party and on facts that existed before or during the audit being conducted. Defimoon is not responsible for any outcome, including changes done to the contract/contracts after the audit was published. This audit is fully objective and only discerns what the contract is saying without adding any opinion to it. Defimoon will always publish its findings regardless of the outcome of the findings. The audit only covers the subject areas detailed in this report and unless specifically stated, nothing else has been audited. Defimoon assumes that the provided information and materials were not altered, suppressed, or misleading. This report is published by Defimoon, and Defimoon has sole ownership of this report. Use of this report for any reason other than for informational purposes on the subjects reviewed in this report including the use of any part of this report is prohibited without the express written consent of Defimoon. In instances where an auditor or team member has a personal connection with the audited project, that auditor or team member will be excluded from viewing or impacting any internal communication regarding the specific audit.

Audit Information

Defimoon utilizes both manual and automated auditing approach to cover the most ground possible. We begin with generic static analysis automated tools to quickly assess the overall state of the contract. We then move to a comprehensive manual code analysis, which enables us to find security flaws that automated tools would miss. Finally, we conduct an extensive unit testing to make sure contract behaves as expected under stress conditions.

In our decision making process we rely on finding located via the manual code inspection and testing. If an automated tool raises a possible vulnerability, we always investigate it further manually to make a final verdict. All our tests are run in a special test environment which matches the "real world" situations and we utilize exact copies of the published or provided contracts.

While conducting the audit, the Defimoon security team uses best practices to ensure that the reviewed contracts are thoroughly examined against all angles of attack. This is done by evaluating the codebase and whether it gives rise to significant risks. During the audit, Defimoon assesses the risks and assigns a risk level to each section together with an explanatory comment.

Contents

Syndiqate token contract

Contract information

Name	Syndiqate
Description	Syndiqate - club-based crypto insurance community offering the market insurance products built on a multi-platform software, using blockchain, smart contracts & NFT technologies. Investment Protection Against Asset Impairment Risk & Scam Risk. Syndiqate's digital

Name	cryptographically secured token SQAT Syndiqate
Website	https://syndiqate.io
Twitter	https://twitter.com/SyndiqateInfo
Contact (Telegram)	https://t.me/syndiqate_chat
Token Name	Syndiqate Token
Token Short	SQAT
Total Supply	1000000000
Token Decimals	18
Contract address	0xd286480b95B0A48805B5f3FeF99883eD828f2ef7
Code Language	Solidity
Chain	BSC

Audit overview

No security issues were found

The contract has no security issues. Small and simple, without complicated mechanics. It is recommended to cover it with tests before any increase in project complexity, so it can be maintained with ease. Please find the full report below for a complete understanding of the audit.

Application security checklist

Test	Result
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

Test	Result
DoS with block gas limit	Passed
Methods execution permissions	Passed
Economy model of the contract	Not Checked
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	Not Checked
Oracle Calls	Not Checked
Cross-function race conditions	Passed
Safe OpenZeppelin contracts and implementation usage	Passed
Whitepaper-Website-Contract correlation	Not checked
Front Running	Not checked

Detailed audit information

Contract Programming

Test	Result
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

Test	Result
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

Test	Result
Visibility not explicitly declared	Passed
Variable storage location not explicitly declared	Passed
Use keywords/functions to be deprecated	Passed
Other code specification issues	Passed

Gas Optimization

Test	Result
Assert () misuse	Passed
High consumption ‘for/while’ loop	Passed
High consumption ‘storage’ storage	Passed
“Out of Gas” Attack	Passed

Business Risk

Test	Result
“Short Address” Attack	Passed
“Double Spend” Attack	Passed

Executive Summary

**According to the standard audit assessment, client's solidity smart contract (syndicate-ico) is well-secured. The contract has successfully passed the audit. No issues were found.

Code Quality

Syndicate-ico project is well coded, the code is clear, concise and follows the best coding practices. Contract is exceptionally well documented. However, it is always recommended to cover your code with tests, which helps to control contract execution in various circumstances

Tests

As mentioned above, it’s recommended to write tests for the smart contract, so that potential vulnerabilities can be fixed at earliest stage. The code was audited from the syndicate-ico project GitHub repository, provided by the client.

Contract overview

Privileged executables

- `grantRole`
- `revokeRole`
- `pause`
- `unpause`

Executables

- `getContractManagers`
- `grantManagerToContractInit`
- `revokeManagerAfterContractInit`
- `_beforeTokenTransfer`

Features

```
function grantRole(bytes32 role, address account) public override onlyRole(getRoleAdmin(role)) {

    uint256 index = contractManagers.length;

    if (role == MANAGER_ROLE) {

        require(isManager[account] == 0, "Account already a manager!");

        contractManagers.push(account);

        isManager[account] = index;

    }

    _grantRole(role, account);

}
```

Privileged Allows an address (ico contract) to manage tokens

```
function revokeRole(bytes32 role, address account) public override onlyRole(getRoleAdmin(role)) {

    uint256 index = isManager[account];

    if (role == MANAGER_ROLE) {

        require(index > 0, "Account not a manager!");

        delete contractManagers[index];

        isManager[account] = 0;

    }

    _revokeRole(role, account);

}
```

Privileged Revokes privileged access from address

```
function pause() public onlyRole(DEFAULT_ADMIN_ROLE) {

    _pause();

}
```

Privileged Triggers stopped state

```
function unpause() public onlyRole(DEFAULT_ADMIN_ROLE) {

    _unpause();

}
```

Privileged Returns to normal state

```
function getContractManagers() public view returns(address[] memory) {

    return(contractManagers);

}
```

Returns privileged addresses that can call this contract

```
function grantManagerToContractInit(address account, uint256 amount) external {

    uint256 index = contractManagers.length;

    require(hasRole(DEFAULT_ADMIN_ROLE, tx.origin) == true, "Caller is not admin!");

    require(isManager[account] == 0, "Account already a manager!");

    _approve(address(this), account, amount);

    contractManagers.push(account);

    isManager[account] = index;

    _grantRole(MANAGER_ROLE, account);

}
```

Allows certain addresses (ico contracts) to recieve tokens at init to set up ICO fund

---

```
function revokeManagerAfterContractInit(address account) external {

    uint256 index = isManager[account];

    require(hasRole(DEFAULT_ADMIN_ROLE, tx.origin) == true, "Caller is not admin!");

    require(index > 0, "Account not a manager!");

    delete contractManagers[index];

    isManager[account] = 0;

    _revokeRole(MANAGER_ROLE, account);

}
```

Revokes the ability to spend tokens after init

```
function _beforeTokenTransfer(address from, address to, uint256 amount)

                                internal

                                whenNotPaused

                                override

{

    super._beforeTokenTransfer(from, to, amount);

}
```

Checks whether the contract is paused before token transfers

Imported contracts and dependencies

Contract/Library/Interfacee	Description
ERC20.sol	Implementation of the industry standard {IERC20} interface.
ERC20Burnable.sol	Extension of {ERC20} that allows token holders to destroy both their own tokens and those that they have an allowance for, in a way that can be recognized off-chain (via event analysis)
Pausable.sol	Contract module which allows children to implement an emergency stop mechanism that can be triggered by an authorized account
AccessControl.sol	Contract module that allows children to implement role-based access control mechanisms

Seed Round contract

Contract information

Name	Syndiqate
Description	Syndiqate - club-based crypto insurance community offering the market insurance products built on a multi-platform software, using blockchain, smart contracts & NFT technologies. Investment Protection Against Asset Impairment Risk & Scam Risk. Syndiqate's digital cryptographically secured token SQAT
Website	https://syndiqate.io
Twitter	https://twitter.com/SyndiqateInfo

Contact Name (Telegram)	Syndiqate https://t.me/syndiqate_chat
Token Name	Syndiqate Token
Token Short	SQAT
Total Supply	1000000000
Token Decimals	18
Contract address	0x51c9bf4A62085acf0feE99cb803c7a3599F353D0
Code Language	Solidity
Chain	BSC

Audit overview

No security issues were found

The contract has no security issues. Small and simple, without complicated mechanics. It is recommended to cover it with tests before any increase in project complexity, so it can be maintained with ease. Please find the full report below for a complete understanding of the audit.

Application security checklist

Test	Result
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	Not Checked

Test	Result
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	Not Checked
Cross-function race conditions	Passed
Safe OpenZeppelin contracts and implementation usage	Passed
Whitepaper-Website-Contract correlation	Not checked
Front Running	Not checked

Detailed audit information

Contract Programming

Test	Result
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

Test	Result
Race condition	Passed
Logical vulnerability	Passed
Other programming issues	Passed

Code Specification

Test	Result
Visibility not explicitly declared	Passed
Variable storage location not explicitly declared	Passed
Use keywords/functions to be deprecated	Passed
Other code specification issues	Passed

Gas Optimization

Test	Result
Assert () misuse	Passed
High consumption ‘for/while’ loop	Passed
High consumption ‘storage’ storage	Passed
“Out of Gas” Attack	Passed

Business Risk

Test	Result
“Short Address” Attack	Passed
“Double Spend” Attack	Passed

Executive Summary

**According to the standard audit assessment, client's solidity smart contract (syndicate-ico) is well-secured. The contract has successfully passed the audit. No issues were found. **

Code Quality

Syndicate-ico project is well coded, the code is clear, concise and follows the best coding practices. Contract is exceptionally well documented. However, it is always recommended to cover your code with tests, which helps to control contract execution in

various circumstances.

Tests

As mentioned above, it's recommended to write tests for the smart contract, so that potential vulnerabilities can be fixed at earliest stage. The code was audited from the syndicate-ico project GitHub repository, provided by the client.

Contract overview

Privileged executables

- `withdrawRaisedFunds`
- `withdrawRemainingSqat`

Executables

- `claimTokens`
- `buySqat`
- `_lockAndDistribute`
- `checkIfActive`

Features

```
function withdrawRaisedFunds(address _reciever) public onlyOwner {

    uint256 balance = USDT.balanceOf(address(this));

    USDT.transfer(_reciever, balance);

}
```

Privileged Allows to withdraw raised funds (USDT)

```
function withdrawRemainingSqat(address _reciever) public onlyOwner ifInactive {

    SQAT.transfer(_reciever, availableTreasury);

    availableTreasury = 0;

}
```

Privileged Allows to withdraw SQAT remaining after the round end

---

```
function claimTokens() public checkLock() {

    address user = msg.sender;

    User storage userStruct = users[user];

    uint256 amountToClaim; // 15%

    if (userStruct.numUnlocks < 5) {

        amountToClaim = (userStruct.pendingForClaim / 10000) * 1500;

    }

    else if (userStruct.numUnlocks == 5) {

        amountToClaim = userStruct.pendingForClaim;

    }

    else {

        revert("Everything is already claimed!");

    }

    SQAT.transfer(user, amountToClaim);

    userStruct.liquidBalance += amountToClaim;

    userStruct.pendingForClaim -= amountToClaim;

    userStruct.nextUnlockDate += LOCK_PERIOD;

    userStruct.numUnlocks += 1;

    userStruct.isLocked = true;

    emit SqatClaimed(user,

    amountToClaim,

    6 - userStruct.numUnlocks,

    userStruct.nextUnlockDate);

}
```

Checks if tokens are unlocked and transfers 15% from pendingForClaim user will recieve all remaining tokens with the last (6th) claim

```
function buySqat(uint256 _amount) public areTokensAvailable(_amount) ifActive {

    address user = msg.sender;

    uint256 priceUSDT = _amount / SQAT_PRICE_USDT;

    require(USDT.balanceOf(user) >= priceUSDT, "Not enough USDT tokens!");

    require(USDT.transferFrom(user, address(this), priceUSDT) == true, "Failed to transfer USDT!");

    _lockAndDistribute(_amount);

    emit SqatPurchased(msg.sender, _amount);

}
```

Allows to purchase SQAT tokens

```
function _lockAndDistribute(uint256 amount) private {

    address user = msg.sender;

    User storage userStruct = users[user];

    uint256 timestampNow = block.timestamp;

    uint256 immediateAmount = (amount / 10000) * 1000; // 10%

    SQAT.transfer(user, immediateAmount); // issue 10% immediately

    if (users[user].totalSqatBalance == 0) {

        icoTokenHolders.push(user);

    }

    userStruct.totalSqatBalance += amount;

    availableTreasury -= amount;

    userStruct.liquidBalance += immediateAmount; // issue 10% immediately to struct

    userStruct.pendingForClaim += amount - immediateAmount; // save the rest

    userStruct.nextUnlockDate = timestampNow + LOCK_PERIOD; // lock for 3 months

    userStruct.isLocked = true;

    userStruct.numUnlocks = 0;

}
```

When user buys SQAT, 10% is issued immediately remaining tokens are locked for 6 * LOCK_PERIOD = 18 months

---

```
function checkIfActive() public returns(bool) {

    if ((block.timestamp <= ROUND_START_DATE) || (block.timestamp >= ROUND_END_DATE) || availableTreas

        isActive = false;

    }

    if (block.timestamp > ROUND_START_DATE && block.timestamp < ROUND_END_DATE && availableTreasury > 

        isActive = true;

    }

    return(isActive);

}
```

Checks if round still active

Imported contracts and dependencies

Contract/Library/Interfacee	Description
Ownable.sol	Contract module which provides a basic access control mechanism, where there is an account (an owner) that can be granted exclusive access to specific functions
IERC20.sol	Interface of the ERC20 standard as defined in the EIP.
ISQAT.sol	SQAT token contract interface

Conclusion

The codebase of the syndicate-ico has passed the audit successfully and can be considered a "Well-Secured" application. The code is well-written, clear and follows the best security practices.

However, due to the nature of the application and given the risks connected with the decentralized finance, we can provide no guarantees on the future outcomes and project operation. We have used all the latest static tools and manual analysis to cover the greatest possible amount of test cases. Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart contracts high-level description of functionality and security was presented in the Application security checklist section of the report.

Audit report contains all found security vulnerabilities and other issues found in the reviewed code.

Security status of the reviewed codebase is "Well-Secured".

Conclusion

The codebase of the syndicate-ico has passed the audit successfully and can be considered a "Well-Secured" application. The code is well-written, clear and follows the best security practices.

However, due to the nature of the application and given the risks connected with the decentralized finance, we can provide no

guarantees on the future outcomes and project operation. We have used all the latest static tools and manual analysis to cover the greatest possible amount of test cases. Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart contracts high-level description of functionality and security was presented in the Application security checklist section of the report.

Audit report contains all found security vulnerabilities and other issues found in the reviewed code.

Security status of the reviewed codebase is "Well-Secured".

Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goal of our security audits is to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing 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 include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high-level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, review open issue tickets, and investigate details other than the implementation.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system to make a final decision.

Suggested Solutions

We suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Appendix A — Risk Ratings

Risk Rating	Description
High Risk	A fatal vulnerability that can cause the loss of all Tokens / Funds.
Medium Risk	A vulnerability that can cause the loss of some Tokens / Funds.
Low Risk	A vulnerability which can cause the loss of protocol functionality.

Risk Rating	Description
Informational	Non-security issues such as functionality, style, and convention.

Appendix B — Finding Statuses

Status	Description
Closed	Contracts were modified to permanently resolve the finding.
Mitigated	The finding was resolved by other methods such as revoking contract ownership. The issue may require monitoring, for example in the case of a time lock.
Partially Closed	Contracts were updated to fix the issue in some parts of the code.
Partially Mitigated	Fixed by project specific methods which cannot be verified on chain. Examples include compounding at a given frequency.
Acknowledged	Project team is made aware of the finding.
Open	The finding was not addressed.