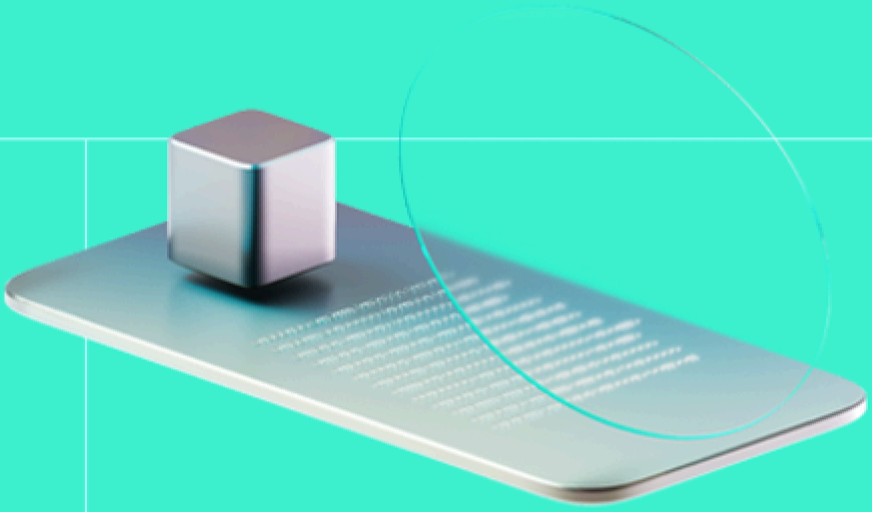
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: ton-club.com

Date: 21.11.2024

We express our gratitude to the Umoja team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

The Umoja Synth Protocol enables users to create automated asset management positions represented on-chain as NFTs (initially non-transferable).
The user deposits collateral via the “UmojaSynthPool.sol” contract where it is moved to an off-chain exchange CEX and managed using our proprietary algorithm. Position NFT data as well as accounting is managed off-chain. In addition, the Synth System also includes an investment pool

 InvestmentPool.sol), where excess
on-chain collateral is deposited into the aave protocol to accrue yield, as well as an insurance pool
 UmojaInsurance.sol) which acts as an emergency fund as well as a source of yield for insurance providers.
Platform: EVM Language: SOLIDITY Tags: NFT, VAULT, ERC4626, EIP712, LENDING, AAVE. Timeline:

16.05.2024   31.05.2024 Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://git hub.com/Ton-Club/Repo/Staking
Commit	d860dc1

Audit Summary

10/10	10/10	99%	10/10
Security score	Code quality score	Test coverage	Documentation quality score

Total 10/10

The system users should acknowledge all the risks summed up in the risks section of the report

0	0	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	0
High	0
Medium	0
Low	0

Document

Name	Smart Contract Code Review and Security Analysis Report for Ton Club
Audited By	David Camps Novi, Viktor Lavrenenko
Approved By	Przemyslaw Swiatowiec
Website	https://ton-club.git book.io/toncly-protocol
Changelog	21/05/2024 Preliminary Report; 31/05/2024 Final Report;

Table of Contents

System Overview

Umoja is an asset management protocol that uses *'smartcoins'* to automate money, enabling money itself to autonomously minimize risk & optimize returns.

It consists of the following contracts:

- src/Errors/Errors.sol - the contract that contains the errors used in the project.
- src/UmojaSynth/InvestmentPool.sol -this contract is used for holding excess collateral USDC by the UmojaSynthPool.
- src/UmojaSynth/synthNFT.sol - this contract is used to mint Synths from the UmojaSynthPool.
- src/UmojaSynth/UmojaInsurance.sol - this contract is a token vault that acts as an insurance pool in case of emergency.
- src/UmojaSynth/UmojaSynthPool.sol - this contract is used to create Synths (on-chain derivative products).
- src/interfaces/ISynthNFT.sol - the interface which stores the template for the synthNFT.sol
- src/interfaces/IUmojaSynthPool.sol - the interface which stores the template for the UmojaSynthPool.sol

Privileged roles

- the contracts in the scope have three roles: the owner, admin, and the SynthPool:
  - InvestmentPool.sol
    - the owner can change the admin and change the reward ratio.
    - the admin can invest funds in the Aave pool, withdraw profits from the Aave pool, withdraw tokens from the Aave pool.
  - SynthNFT.sol
    - the SynthPool can mint a new position, close the position and burn the position.
    - the owner can change the baseURI of the NFT and stop the transfer of tokens via switchTransferAllowed().
  - UmojaSynt hPool.sol
    - the owner can set the address of the NFT contract.
    - the owner can withdraw ERC20 tokens.
    - the owner can make a partial refund.
    - the owner of the NFT can topup position, request a refund and close the position,
  - UmojaInsurance.sol
    - the owner can change the withdrawalFee, change the reward period and send the insurance fund in a case of emergency.

Executive Summary

This report presents an in-depth analysis and scoring of the customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

The total Documentation quality score is 10 out of 10.

- Functional requirements are complete.
- Technical description is complete.

Code quality

The total Code quality score is 10 out of 10.

- Best practices are followed.
- The development environment is configured.

Test coverage

Code coverage of the project is 99% (branch coverage).

- Deployment and basic user interactions are covered with tests.

Security score

Upon auditing, the code was found to contain 0 critical, 0 high, 0 medium, and 0 low severity issues. Out of these, 0 issues have been addressed and resolved, leading to a security score of 10 out of 10.

All identified issues are detailed in the “Findings” section of this report.

Summary

The comprehensive audit of the customer's smart contract yields an overall score of 10. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- The owner of the UmojaSynthPool can withdraw any amount of collateral tokens via UmojaSynthPool::withdrawToken(), which can lead to security incidents due to the high level of centralization. A similar scenario is present in UmojaInsurance::sendInsuranceFunds() and InvestmentPool::withdrawToken().
- The project utilizes Solidity version 0.8.23, which includes the introduction of the PUSHO 0 5f) opcode. This opcode is currently supported on the Ethereum mainnet but may not be universally supported across other blockchain networks. Consequently, deploying the contract on chains other than the Ethereum mainnet, such as certain Layer 2 L2 chains or alternative networks, might lead to compatibility issues or execution errors due to the lack of support for the PUSHO opcode. In scenarios where deployment on various chains is anticipated, selecting an appropriate Ethereum Virtual Machine EVM version that is widely supported across these networks is crucial to avoid potential operational disruptions or deployment failures.
- The owner of the UmojaSynthPool is responsible for refunding the tokens back to the users. There is a risk that the users will not receive their assets if the owner lacks the funds.
- It is recommended to use the multisig wallet for the owner and the admin roles. Otherwise, it can lead to security problems if the owner's or admin's address becomes malicious.
- The InvestmentPool.sol lacks the setter functions for the critical addresses including aavePool, which creates a security risk if the Aave's address will be changed in the future.
- The project's contracts lack a pause feature. This functionality can be used to manage the protocol in case any issue arises that require the lock down of the system .
- The project is fully or partially centralized, introducing single points of failure and control. This centralization can lead to vulnerabilities in decision-making and operational processes, making the system more susceptible to targeted attacks or manipulation.
- The digital contract architecture relies on administrative keys for critical operations. Centralized control over these keys presents a significant security risk, as compromise or misuse can lead to unauthorized actions or loss of funds.
- This protocol interacts with AAVE. Dependence on external DeFi protocols inherits their risks and vulnerabilities. This might lead to direct financial losses if these protocols are exploited, indirectly affecting the audited project.
- The project's contracts are upgradable, allowing the administrator to update the contract logic at any time. While this provides flexibility in addressing issues and evolving the project, it also introduces risks if upgrade processes are not properly managed or secured, potentially allowing for unauthorized changes that could compromise the project's integrity and security.

Findings

Vulnerability Details

F-2024-3060 - The usage of the precompile ecrecover can lead to signature mailability - Medium

Description:

The functions UmojaSynthPool::getSigner() and UmojaSynthPool::getTopUpSigner() are be vulnerable to a signature malleability attack.

This vulnerability stems from the function's inability to discern between legitimately unique signatures and those that have been manipulated but are still considered valid by the Ethereum blockchain's signature verification standards. By exploiting this flaw, an external actor can create signatures that will be accepted by the system, enabling unauthorized transactions.
The vulnerable functions can be found in the code snippet below:

```
/**
* @notice recovers the signer address from SynthData struct param and v, r, s ent
* @dev called from "CreatePosition
* @param data struct containing position details (user address, collateral amount
* @param v signature param
* @param r signature param
* @param s signature param
* @return signer address of signer. Must match "transactionSigner"
*/
function getSigner(
    SynthData memory data,
    uint8 v,
    bytes32 r,
    bytes32 s
) public view virtual override returns (address signer) {
    signer = ecrecover(getTypedDataHash(data), v, r, s);


/**
* @notice recovers the signer address from SynthData Topup struct param and v, r
* @dev called from "topupPosition"
* @param data struct containing topup details (user address, collateral topup amo
* @param v signature param
* @param r signature param
* @param s signature param
* @return signer address of signer. Must match "transactionSigner"
```

```
*/
function getTopUpSigner(
        TopUpData memory data,
        uint8 v,
        bytes32 r,
        bytes32 s
) public view virtual override returns (address signer) {
        signer = ecrecover(getTypedTopUpDataHash(data), v, r, s);
}
```

Assets:

- UmojaSynth/UmojaSynthPool.sol [https://github.com/Umoja-Labs/hedge-protocol/tree/finalChanges]

Status: Fixed

Classification

Impact: 4/5 3/5 Independent

Likelihood: Medium Likelihood 1 5 3

Exploitability: Impact 1 5 4

Complexity: Exploitability 0 2 0

Complexity 0 2 1 Final

Score: 3.3 Medium)

Severity: Medium

Recommendations

Remediation: To enhance the security of your Solidity smart contracts and mitigate the risk of signature malleability attacks, it is advisable to use OpenZeppelin's ECDSA library instead of the built-in ecrecover() function. The ECDSA library provides robust and reliable signature verification, reducing the vulnerability to replay attacks and ensuring the integrity of the contract interactions.

Resolution: Resolved in commitID b464b78: ECDSA library was used to recover the signer.

F-2024-3085 - Missing zero address check may result in loss of funds - Low

Description:

The treasuryAddr input parameter in initialize() is missing a zero address check. Given that funds are sent to such address, it is critical to make sure it is properly setup to avoid loss of funds.

```
function initialize(address adminAddr, address usdc, address pool, address aUsdcA
    initializer
    notZeroAddress(adminAddr)
    notZeroAddress(usdc)
    notZeroAddress(pool)
    notZeroAddress(aUsdcAddr)
    notZeroAddress(insurance)
{
    __Ownable_init(msg.sender);
    usdcAddress = usdc;
    admin = adminAddr;
    aavePool = pool;
    aUSDC = aUsdcAddr;
    insurancePool = insurance;
    treasury = treasuryAddr;
    rewardRatio = 500;

    emit NewAdmin(admin);
}

function withdrawProfits() public isAdmin {
    uint256 aUsdcBalance = checkATokenBalance();
    uint256 profit = aUsdcBalance - principleAmount;
    if (profit == 0) {
        revert Errors.INSUFFICIENT_PROFIT();
    }
    uint256 insuranceShare = profit * rewardRatio / 1000;
    IERC20 USDC = IERC20(usdcAddress);
    IPool(aavePool).withdraw(usdcAddress, profit, address(this));
    USDC.safeTransfer(insurancePool, insuranceShare);
    USDC.safeTransfer(treasury, profit - insuranceShare);

    emit WithdrawProfit(profit);
}
```

In Solidity, the Ethereum address 0x0000000000000000000000000000000000000000 is known as the "zero address". This address has significance because it is the default value for uninitialized address variables and is often used to represent an

invalid or non-existent address. The "Missing zero address control" issue arises when a Solidity smart contract does not properly check or prevent interactions with the zero address, leading to unintended behavior.
For instance, a contract might allow tokens to be sent to the zero address without any checks, which essentially burns those tokens as they become irretrievable. While sometimes this is intentional, without proper control or checks, accidental transfers could occur.

Assets:	• UmojaSynth/InvestmentPool.sol [https://github.com/Umoja-Labs/hedge-protocol/tree/finalChanges]
Status:	Fixed

Classification

Impact:	2/5 2/5 Likelihood
Likelihood:	1 5 2 Impact 1 5 2 Exploitability 1 2 1 Complexity 0 2 0 Final Score: 2.0 (Low)
Severity:	Low

Recommendations

Remediation:	Consider adding a zero address check for `treasuryAddr` in `initialize()` functions.
Resolution:	Fixed in commitID b464b78 : a zero address check was introduced for `treasuryAddr`.

Observation Details

F-2024-3000 - Missing SPDX license identi er - Info

Description:	InvestmentPool, synthNFT and UmojaSynthPool contracts lack an SPDX License-Identifier, a key component for specifying the license under which the contract is released. This identifier is essential for informing users and developers of the legal permissions and restrictions applied to the code. Following best practices, including an SPDX License-Identifier, is a standard in the development of smart contracts. The absence of an SPDX License-Identifier can lead to legal ambiguity regarding the usage, modification, and distribution of the contract's code. It is crucial for open-source software and smart contracts to clearly state their licensing terms to ensure compliance with legal requirements and to promote transparency in the blockchain community.
Assets:	• UmojaSynth/InvestmentPool.sol [https://github.com/Umoja-Labs/hedge-protocol/tree/finalChanges] • UmojaSynth/synthNFT.sol [https://github.com/Umoja-Labs/hedge-protocol/tree/finalChanges] • UmojaSynth/UmojaSynthPool.sol [https://github.com/Umoja-Labs/hedge-protocol/tree/finalChanges]
Status:	Fixed

Recommendations

Remediation:	Include an SPDX License-Identifier at the beginning of the assets. Choose a license that aligns with the project’s goals and legal requirements.
Resolution:	Fixed in commitID b464b78: `// SPDX-License-Identifier: Apache-2.0` was added in the reported contracts.

F-2024-3017 - OwnableUpgradeable uses single-step ownership transfer pa ern - Info

Description:	The contracts in the scope currently use simple OwnableUpgradeable pattern, where ownership can be transferred in a single transaction. While this is straightforward to understand, it can potentially lead to issues if the new owner's address is input incorrectly, as ownership would be irreversibly transferred to an incorrect (and potentially inaccessible) address. Ownable2StepUpgradeable prevents the contract ownership from mistakenly being transferred to an address that cannot handle it, by requiring that the recipient of the owner permissions actively accept via a contract call of its own.
Assets:	• UmojaSynth/InvestmentPool.sol [https://github.com/Umoja-Labs/hedge-protocol/tree/finalChanges] • UmojaSynth/synthNFT.sol [https://github.com/Umoja-Labs/hedge-protocol/tree/finalChanges] • UmojaSynth/UmojaInsurance.sol [https://github.com/Umoja-Labs/hedge-protocol/tree/finalChanges] • UmojaSynth/UmojaSynthPool.sol [https://github.com/Umoja-Labs/hedge-protocol/tree/finalChanges]
Status:	Fixed

Recommendations

Remediation:	Consider using Ownable2StepUpgradeable from OpenZeppelin Contracts to enhance the security of your contract ownership management. These contracts prevent the accidental transfer of ownership to an address that cannot handle it, such as due to a typo, by requiring the recipient of owner permissions to actively accept ownership via a contract call. This two-step ownership transfer process adds an additional layer of security to your contract's ownership management.
Resolution:	Fixed in commitID b464b78: OwnableUpgradeable was updated to Ownable2StepUpgradeable.

F-2024-3059 - Missing events emi ing for critical functions - Info

Description:	Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes with timelocks that allow users to evaluate them and consider if they would like to engage/exit based on how they perceive the changes as affecting the trustworthiness of the protocol or profitability of the implemented financial services. The alternative of directly querying the on-chain contract state for such changes is not considered practical for most users/usages. Missing events do not promote transparency and if such changes immediately affect users' perception of fairness or trustworthiness, they could exit the protocol causing a reduction in liquidity which could negatively impact protocol TVL and reputation. The following functions that do not emit any events in the following functions: • UmojaInsurance: changeWithdrawalFee(), changeRewardPeriod() • SynthNFT: switchTransferAllowed()
Assets:	• UmojaSynth/synthNFT.sol [https://github.com/Umoja-Labs/hedge-protocol/tree/finalChanges] • UmojaSynth/UmojaInsurance.sol [https://github.com/Umoja-Labs/hedge-protocol/tree/finalChanges]
Status:	Fixed

Recommendations

Remediation:	Consider emitting the corresponding events in the reported functions.
Resolution:	Fixed in commitID b464b78: events were added in the reported functions.

F-2024-3061 - Checks-e ects-interactions pa ern violation - Info

Description: In order to comply with the checks-effects-interactions pattern when depositing funds, the token transfer should be performed before updating the state variables.

The following functions do not follow the aforementioned pattern: UmojaSynthPool::createPosition().

```
function createPosition(
    SynthData memory data,
    uint8 v,
    bytes32 r,
    bytes32 s
) external override nonReentrant {
    ...

    usedNonces[data.nonce] = true;

    uint256 tokenID = synthPositionNFT.mintPosition(msg.sender);

    paymentToken.safeTransferFrom(
        msg.sender,
        address(this),
        data.collateral
    );

    ...
}
```

Assets:

- UmojaSynth/UmojaSynthPool.sol [https://github.com/Umoja-Labs/hedge-protocol/tree/finalChanges]

Status: Fixed

Recommendations

Remediation: It is recommended to update state variables after performing the token transfer.

Resolution: Fixed in commitID b464b78 : the token deposit was moved before the NFT mint.

F-2024-3087 - Potential NFT stuck when using _mint() instead of _safeMint() - Info

Description: The SynthNFT::mintPosition() function currently utilizes the _mint() method for creating new NFTs. However, this approach lacks the safety checks provided by _safeMint(), particularly when the recipient is a contract. The absence of these checks can lead to the loss of NFT if the recipient contract is not designed to handle NFTs. The vulnerable code can be found in the code snippet below:

```
/**
     * @notice mints an NFT.
     * @dev can only be called from Umoja Synth Pool
     * @param user recipient of the minted synth NFT
     * @return _tokenID ID of newly minted synth NFT
     */
    function mintPosition(
        address user
    )
        external
        override
        notZeroAddress(user)
        isSynthPool
        returns (uint256 _tokenID)
    {
        _tokenID = nextID;
        _mint(user, _tokenID);
        nextID++;
        emit NewPosition(_tokenID, user);
    }
```

Assets:

- UmojaSynth/synthNFT.sol [https://github.com/Umoja-Labs/hedge-protocol/tree/finalChanges]

Status: Fixed

Recommendations

Remediation: It is recommended to utilize the `_safeMint()` function instead of _mint() in the `mintPosition()`.

Resolution:	Fixed in commitID b464b78: `_safeMint()` was introduced in place of `_mint()`.

F-2024-3194 - Improper handling of non-compliant ERC20 tokens - Info

Description:

The function InvestmentPool::invest() deposits collateral into lending protocol Aave. Before the transfer of assets, it gives allowance to Aave's Pool contract using the IERC20.approve() function. It can be seen from the code snippet below.

```
function invest(uint256 amount) public isAdmin {
        address pool = aavePool;
        IERC20 USDC = IERC20(usdcAddress);
        principleAmount += amount;
        USDC.approve(pool, amount);
        IPool(pool).supply(
                address(USDC),
                amount,
                address(this),
                0
        );
        emit DepositToLendingProtocol(amount);
}
```

However, the IERC20().approve() brings the risk that the calls to tokens that do not follow the ERC20 standard and lack the return value in their approve() function, are assumed to be successful. As a consequence, the revert scenarios might not be properly handled, which will not lead to the halt of the transaction.

Assets:

- UmojaSynth/InvestmentPool.sol [https://github.com/Umoja-Labs/hedge-protocol/tree/finalChanges]

Status: Fixed

Recommendations

Remediation:

Consider using forceApprove() method from OpenZeppelin's SafeERC20.sol library to ensure that non-compliant ERC20 tokens are handled properly and the transaction reverts in case of failure.

Resolution:

Fixed in commitID b464b78: `approve()` was updated to `forceApprove()`.

F-2024-3195 - Best Practice Violation: Non Upgradeable ReentrancyGuard Inheritance - Info

Description:	Upgradeable contracts that inherit from OpenZeppelin's ReentrancyGuard contract should use its upgradeable version: ReentrancyGuardUpgradeable. This happens because `ReentrancyGuard` contract will set the status to NOT_ENTERED once the contract is deployed, but initializable contracts need to do this step during initialization, not deployment.
Assets:	• UmojaSynth/UmojaInsurance.sol [https://github.com/Umoja-Labs/hedge-protocol/tree/finalChanges] • UmojaSynth/UmojaSynthPool.sol [https://github.com/Umoja-Labs/hedge-protocol/tree/finalChanges]
Status:	Fixed

Recommendations

Remediation:	Replace `ReentrancyGuard` with `ReentrancyGuardUpgradeable` and call `ReentrancyGuardUpgradeable.__ReentrancyGuard_init()` during initialization.
Resolution:	Fixed in commitID b464b78: ReentrancyGuard was updated to ReentrancyGuardUpgradeable in the reported contracts. A call to `ReentrancyGuardUpgradeable.__ReentrancyGuard_init()` was introduced into the initialization of such contracts.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.
While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity De nitions

When auditing smart contracts, Hacken is using a risk-based approach that considers Likelihood, Impact, Exploitability and Complexity metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Contracts in Scope

src/Errors/Errors.sol

src/UmojaSynth/InvestmentPool.sol

src/UmojaSynth/synthNFT.sol

src/UmojaSynth/UmojaInsurance.sol

src/UmojaSynth/UmojaSynthPool.sol

src/interfaces/ISynthNFT.sol

src/interfaces/IUmojaSynthPool.sol