Competitive Security Assessment

KaratDAO

May 4th, 2023

Secure3

secure3.io

Summary

Karat Network is a decentralized data identity protocol that allows users to take control of their Web2 and Web3 data while earning rewards for participating in the network. It is the only Data Middleware that is fully on-chain and provides users with the ability to share and control their data. The network opens up limitless possibilities such as personal data marketplace and encourages dApps to build on top of it.

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

- Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
- Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
- Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
- Verify the code base is compliant with the most up-to-date industry standards and security best practices.
- Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

Overview

Project Detail

Project Name	KaratDAO
Platform & Language	Solidity
Codebase	• https://github.com/KaratDAO/Karat-Network-Contracts • audit commit - f31a266c2f99c3de34890dd6332a448685fa0a98 • final commit - b8e0f11538500d2cd31b72fd5542fa1ec40c9d59
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

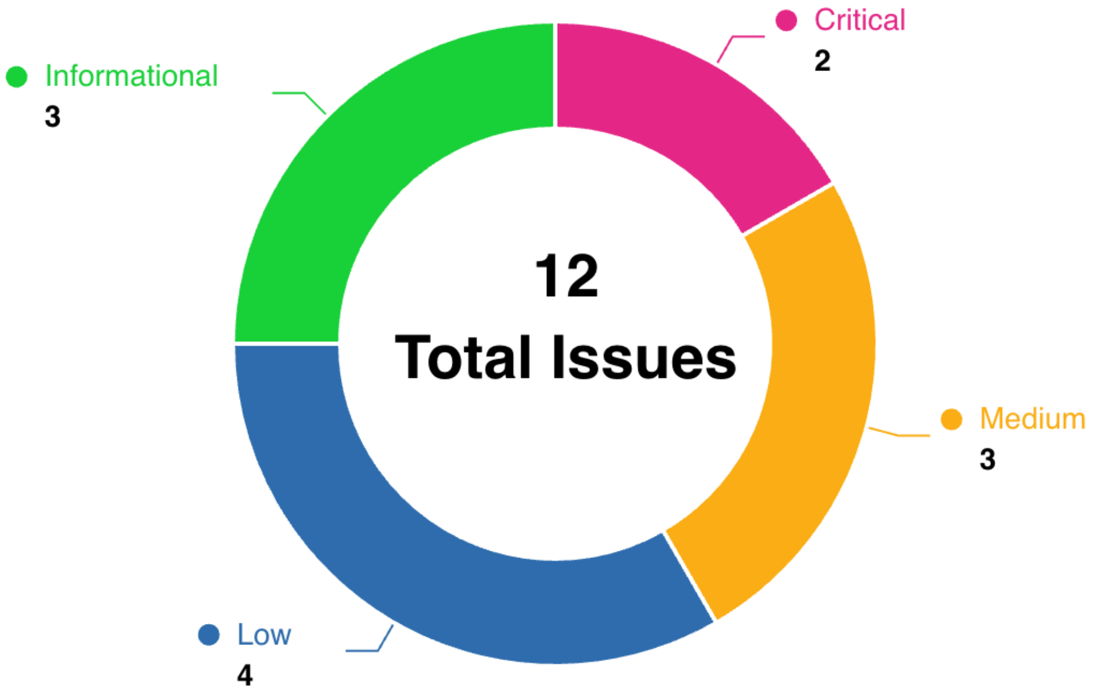
Code Vulnerability Review Summary

Vulnerability Level	Total	Reported	Acknowledged	Fixed	Mitigated	Declined
Critical	**2**	**0**	**0**	**1**	**0**	**1**
Medium	**3**	**0**	**0**	**3**	**0**	**0**
Low	**4**	**0**	**0**	**4**	**0**	**0**
Informational	**3**	**0**	**1**	**2**	**0**	**0**

Audit Scope

File	Commit Hash
contracts/Validator.sol	f31a266c2f99c3de34890dd6332a448685fa0a98
contracts/Claimer.sol	f31a266c2f99c3de34890dd6332a448685fa0a98
contracts/Escrow.sol	f31a266c2f99c3de34890dd6332a448685fa0a98
contracts/IValidator.sol	f31a266c2f99c3de34890dd6332a448685fa0a98
contracts/Proxy.sol	f31a266c2f99c3de34890dd6332a448685fa0a98

Code Assessment Findings



ID	Name	Category	Severity	Status	Contributor
KTD-1	Incorrect require condition in `updateMaxScore`	Logical	Low	Fixed	jayphbee, Hacker007
KTD-2	Missing input validation	Logical	Low	Fixed	Hacker007
KTD-3	Redundancy amount check in mintValidatorPublicBatch	Gas Optimization	Informational	Fixed	LiRiu
KTD-4	Unsafe erc20 transfer methods used	Logical	Medium	Fixed	jayphbee, LiRiu
KTD-5	Unused event AuthorizedSignerUpdated	Gas Optimization	Informational	Fixed	Hacker007

KTD-6	illegal double karatScore for claimer	Logical	Medium	Fixed	LiRiu
KTD-7	incorrect implementation of `_beforeTokenTransfer` function	Logical	Critical	Declined	jayphbee
KTD-8	off-by-one error when validate validatorMaxSupply	Logical	Low	Fixed	jayphbee
KTD-9	omit the roles length check	Logical	Low	Fixed	jayphbee, Hacker007
KTD-10	restrict `msg.value` equal to the exact mint price	Logical	Medium	Fixed	jayphbee, Hacker007
KTD-11	signature replay attack vulnerability in `mintClaimerwithSig` function	Logical	Critical	Fixed	jayphbee, LiRiu
KTD-12	use `abi.encode` instead of `abi.encodePacked`	Gas Optimization	Informational	Acknowledged	LiRiu

KTD-1:Incorrect require condition in `updateMaxScore`

Category	Severity	Code Reference	Status	Contributor
Logical	Low	- code/contracts/Claimer.sol#L59	Fixed	jayphbee, Hacker007

Code

```
59:        require(maxInitialKaratScore >0, "Illegal Score");
```

Description

jayphbee : Incorrect `require` condition in `updateMaxScore`.
Hacker007 : The function `updateMaxScore()` is intended to update the `maxInitialKaratScore`.

```
    function updateMaxScore(uint256 maxScore) public onlyRole(DEFAULT_ADMIN_ROLE) {
        require(maxInitialKaratScore >0, "Illegal Score");

        maxInitialKaratScore = maxScore;
        emit UpdateMaxScore(maxScore);
    }
```

However, the require statement checks if the current value `maxInitialKaratScore` instead of the input parameter `maxScore` is greater than 0. Thus, the admin can succeed in calling the function `updateMaxScore()` with input value 0, which induces the function `_setKaratScore()` to get stuck forever. PoC Using the Foundry to reproduce the issue.

```
    function testUpdateMaxScoreGetStuck() public {
        vm.startPrank(alice);
        nft.updateMaxScore(0);
        console2.log("Start to mint an NFT");
        nft.mintClaimer(address(2), 1, 20, address(3), ClaimerNFT.Role(0));
        vm.stopPrank();
    }
    function testUpdateMaxScoreTwice() public {
        vm.startPrank(alice);
        nft.updateMaxScore(0);
        console2.log("Update again");
        nft.updateMaxScore(100);
        vm.stopPrank();
    }
// Running 2 tests for test/Claimer.t.sol:ClaimerTest
// [FAIL. Reason: Illegal Score] testUpdateMaxScoreGetStuck() (gas: 37772)
// Logs:
//   0x0000000000000000000000000000000000000001
//   Start to mint an NFT

// [FAIL. Reason: Illegal Score] testUpdateMaxScoreTwice() (gas: 23596)
// Logs:
//   0x0000000000000000000000000000000000000001
//   Update again

// Test result: FAILED. 0 passed; 2 failed; finished in 4.89ms

// Failing tests:
// Encountered 2 failing tests in test/Claimer.t.sol:ClaimerTest
// [FAIL. Reason: Illegal Score] testUpdateMaxScoreGetStuck() (gas: 37772)
// [FAIL. Reason: Illegal Score] testUpdateMaxScoreTwice() (gas: 23596)
```

Recommendation

jayphbee :

```
59: require(maxScore >0, "Illegal Score");
```

Hacker007 : Update the input validation logic as below:

```
    function updateMaxScore(uint256 maxScore) public onlyRole(DEFAULT_ADMIN_ROLE) {
        require(maxScore >0, "Illegal Score");

        maxInitialKaratScore = maxScore;
        emit UpdateMaxScore(maxScore);
    }
```

Client Response

Fixed

KTD-2:Missing input validation

Category	Severity	Code Reference	Status	Contributor
Logical	**Low**	• **code/contracts/Claimer.sol#L52-L54**	**Fixed**	**Hacker007**

Code

```
52:        validatorNFT = IValidator(validatorContractAddress);
53:        baseURI = _baseURI;
54:        maxInitialKaratScore = maxKaratScore;
```

Description

Hacker007 : The function `initialize()` does not check if the input parameter `validatorContractAddress` is zero address and the input parameter `maxKaratScore` is greater than zero.

```
        validatorNFT = IValidator(validatorContractAddress);
        baseURI = _baseURI;
        maxInitialKaratScore = maxKaratScore;
```

If `validatorContractAddress` is zero address or `maxKaratScore` is zero, the function `_mintClaimer()` will get stuck and no NFT can be minted.

Recommendation

Hacker007 : Add checks for input parameters `validatorContractAddress` and `maxKaratScore`.

Client Response

Fixed

KTD-3:Redundancy amount check in mintValidatorPublicBatch

Category	Severity	Code Reference	Status	Contributor
Gas Optimization	Informational	- code/contracts/Validator.sol#L181	Fixed	LiRiu

Code

```
181:        require(validatorCounter[tier].current() + amount <= mintBatch[tier], "Exceed Batch Size");
```

Description

LiRiu : In the _mintValidator function it has been checked whether the number of mint exceeds the limit.

```
    //mintValidator NFT in Private Sale with White List or Public Sale
    function _mintValidator(address to, uint256 tier) internal {
        require(validatorCounter[tier].current() < mintBatch[tier], "Exceed batch amount");
        uint256 tokenId = tokenIdCounter.current();
        ...
    }
```

MintValidatorPublicBatch checks the quantity again, wasting Gas

```
    function mintValidatorPublicBatch(address to, uint256 amount, uint256 tier) public payable nonReentrant {
        ...
        require(validatorCounter[tier].current() + amount <= mintBatch[tier], "Exceed Batch Size");
        for (uint256 i = 0; i < amount; i++) {
            _mintValidator(to, tier);
        }
    }
```

Recommendation

LiRiu : remove Validator.sol::L181 check of amount.

Client Response

Fixed

KTD-4:Unsafe erc20 transfer methods used

Category	Severity	Code Reference	Status	Contributor
Logical	Medium	• code/contracts/Escrow.sol#L37 • code/contracts/Escrow.sol#L55	Fixed	jayphbee, LiRiu

Code

```
37:        IERC20(token).transferFrom(msg.sender, address(this), amount);

55:        IERC20(token).transferFrom(msg.sender, address(this), amount);
```

Description

jayphbee : In the `depositERC20` and `depositValidatorERC20` functions, the contract uses `IERC20(token).transferFrom(msg.sender, address(this), amount);` to transfer tokens from the user to the contract. This could be problematic if the token being used doesn't adhere to the ERC20 standard or has a non-standard implementation, for example USDT in mainnet.

LiRiu : In the `Escrow` contract, there are two "transferFrom" function calls that do not check their return values. This introduces a security risk of fund theft.

For example, in the "depositERC20" function,

```
    function depositERC20(address token, uint256 amount) external {
        require(isAuthorizedToken(token), "Invalid token");
        require(amount > 0, "Invalid amount");

        IERC20(token).transferFrom(msg.sender, address(this), amount);
        claimerBalance[msg.sender][token] += amount;
        treasury[token] += amount;
        emit DepositReceived(msg.sender, amount, token);
    }
```

this function updated the variables "claimerBalance" and "treasury" without checking whether "transferFrom" was successful.

These two variables respectively keep track of the deposits of the current user and all users.

The same error occurs in the "depositValidatorToken" function as well.

```
    function depositValidatorERC20(address token, uint256 amount) external {
        require(isAuthorizedToken(token), "Invalid token");
        require(amount > 0, "Invalid amount");

        IERC20(token).transferFrom(msg.sender, address(this), amount);
        validatorBalance[msg.sender][token] += amount;
        treasury[token] += amount;
        emit DepositReceived(msg.sender, amount, token);
    }
```

The ERC20 protocol does not require tokens to revert on transfer errors. Some tokens use a boolean return value in their implementation of "transferFrom" to indicate whether the transfer was successful.

An attacker can increase `claimerBalance`/`validatorBalance` and `treasury` at no cost by deliberately causing "transferFrom" to fail.

Finally, the attacker can steal tokens that do not belong to them by requesting the owner to call the "withdraw" function.

Here's the PoC: SimplifiedEscrow

```
// SPDX-License-Identifier: MIT
pragma solidity 0.5.17;
import "./erc20.sol";
contract SimplifiedEscrowPoC {
    uint256 public amount_token = 0;
    WeakToken public token;

    constructor() public {
        token = new WeakToken();
        token.mint(address(this), 10);
    }

    function depositERC20(address _token, uint256 amount) public {
        ERC20(_token).transferFrom(msg.sender, address(this), amount);
        amount_token += amount;
    }

    function withdraw(uint256 amount) public {
        require(amount > 0, "Invalid amount");
        require(amount_token >= amount, "Insufficient balance");
        (bool success, ) = address(token).call(
                abi.encodeWithSelector(
                    ERC20(token).transfer.selector,
                    msg.sender,
                    amount
                )
            );
        require(success, "Transfer failed");
    }

    function attack() external {
        depositERC20(address(token), 10);
        withdraw(10);
    }
}
```

WeakToken

```
//SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.5.10;

contract ERC20 {

    mapping (address => uint256) private _balances;

    mapping (address => mapping (address => uint256)) private _allowances;

    uint256 private _totalSupply;

    string private _name;
    string private _symbol;
    uint8 private _decimals;

    constructor (string memory name, string memory symbol) public {
        _name = name;
        _symbol = symbol;
        _decimals = 18;
    }

    /**
     * @dev Returns the name of the token.
     */
    function name() public view returns (string memory) {
        return _name;
    }

    function symbol() public view returns (string memory) {
        return _symbol;
    }

    function decimals() public view returns (uint8) {
        return _decimals;
    }

    /**
     * @dev See {IERC20-totalSupply}.
     */
    function totalSupply() public view returns (uint256) {
        return _totalSupply;
    }
```

```
/**
     * @dev See {IERC20-balanceOf}.
     */
    function balanceOf(address account) public view returns (uint256) {
        return _balances[account];
    }

    function transfer(address recipient, uint256 amount) public returns (bool) {
        _transfer(msg.sender, recipient, amount);
        return true;
    }

    function allowance(address owner, address spender) public view returns (uint256) {
        return _allowances[owner][spender];
    }

    function approve(address spender, uint256 amount) public returns (bool) {
        _approve(msg.sender, spender, amount);
        return true;
    }

    function transferFrom(address sender, address recipient, uint256 amount) public returns (bool) {
        _transfer(sender, recipient, amount);
        _approve(sender, msg.sender, _allowances[sender][msg.sender] - amount);
        return true;
    }

    function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
        _approve(msg.sender, spender, _allowances[msg.sender][spender] + addedValue);
        return true;
    }

    function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
        _approve(msg.sender, spender, _allowances[msg.sender][spender] - subtractedValue);
        return true;
    }

    function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "ERC20: transfer from the zero address");
require(recipient != address(0), "ERC20: transfer to the zero address");
        _balances[sender] = _balances[sender] - amount;
        _balances[recipient] = _balances[recipient] + amount;
    }

    function _mint(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: mint to the zero address");
        _totalSupply = _totalSupply + amount;
        _balances[account] = _balances[account] + amount;
    }

    function _burn(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: burn from the zero address");
        _balances[account] = _balances[account] - amount;
        _totalSupply = _totalSupply - amount;
    }
    function _approve(address owner, address spender, uint256 amount) internal {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
    }
    function _setupDecimals(uint8 decimals_) internal {
        _decimals = decimals_;
    }

}

contract WeakToken is ERC20("Weak Token", "WT") {
    function mint(address _to, uint256 _amount) public {
        _mint(_to, _amount);
    }
    function burn(address _from ,uint _amount) public {
        _burn(_from,_amount);
    }
}
```

Recommendation

jayphbee : This can be addressed by using the OpenZeppelin's `SafeERC20` library, which provides a more robust and safer way of interacting with ERC20 tokens.
LiRiu : Use SafeERC20 wrapper using SafeERC20 for ERC20.

Client Response

Fixed

Implemented safeerc20upgradable instead of safeERC20

KTD-5:Unused event AuthorizedSignerUpdated

Category	Severity	Code Reference	Status	Contributor
Gas Optimization	**Informational**	• **code/contracts/Claimer.sol#L20**	**Fixed**	**Hacker007**

Code

```
20:    event AuthorizedSignerUpdated(address indexed signer, bool isAuthorized);
```

Description

Hacker007 : The event `AuthorizedSignerUpdated` is declared but never used in the contract.

Recommendation

Hacker007 : remove the unused event.

Client Response

Fixed

KTD-6:illegal double karatScore for claimer

Category	Severity	Code Reference	Status	Contributor
Logical	Medium	• code/contracts/Validator.sol#L202-L214	Fixed	LiRiu

Code

```
202:    function setReferral(uint256 validatorTokenId, address claimerAddr, address lieutenantAddr, uint256 claimerKaratScore) external {
203:        require(authorizedCallers[msg.sender] == true, "Not Authorized Caller");
204:        require(_exists(validatorTokenId), "Not A Valid Referral");
205:
206:        validatorMap[validatorTokenId][claimerAddr] = true;
207:        validatorRefereeCounter[validatorTokenId]++;
208:        validatorKaratScore[validatorTokenId] += claimerKaratScore;
209:        if (lieutenantAddr != address(0)) {
210:            require(validatorMap[validatorTokenId][lieutenantAddr], "Lieutenant is not associated");
211:            lieutenantMap[lieutenantAddr][claimerAddr] = true;
212:            lieutenantRefereeCounter[lieutenantAddr]++;
213:            lieutenantKaratScore[lieutenantAddr] += claimerKaratScore;
214:        }
```

Description

LiRiu : In the `Claimer` contract, the signature message of mintClaimerwithSig does not include `lieutenantAddr`. At the same time, the `setReferral` function in the `Validator` contract does not check the validity of `lieutenantAddr`.

```
    function setReferral(uint256 validatorTokenId, address claimerAddr, address lieutenantAddr, uint256 claimerKaratScore) external {
        require(authorizedCallers[msg.sender] == true, "Not Authorized Caller");
        require(_exists(validatorTokenId), "Not A Valid Referral");

        validatorMap[validatorTokenId][claimerAddr] = true;
        validatorRefereeCounter[validatorTokenId]++;
        validatorKaratScore[validatorTokenId] += claimerKaratScore;
        if (lieutenantAddr != address(0)) {
            require(validatorMap[validatorTokenId][lieutenantAddr], "Lieutenant is not associated");
            lieutenantMap[lieutenantAddr][claimerAddr] = true;
            lieutenantRefereeCounter[lieutenantAddr]++;
            lieutenantKaratScore[lieutenantAddr] += claimerKaratScore;
        }

        emit ReferralSet(validatorTokenId, claimerAddr, lieutenantAddr, claimerKaratScore);
    }
```

This means that users are free to choose their lieutenant even theirselves.

As a claimer, attacker can set lieutenantAddr to himself to get extra lieutenantKaratScore

Recommendation

LiRiu : require `lieutenantKaratScore` is not equal with `claimerAddr`

Client Response

Fixed

KTD-7:incorrect implementation of `_beforeTokenTransfer` function

Category	Severity	Code Reference	Status	Contributor
Logical	**Critical**	• **code/contracts/Claimer.sol#L135-L139**	**Declined**	**jayphbee**

Code

```
135:    //Only Minter are able to burn/transfer
136:    function _beforeTokenTransfer(address from, address to, uint256 tokenId, uint256 batchSize)
internal virtual override(ERC721EnumerableUpgradeable) {
137:        require(from == address(0), "Tranfer/Burn are not allowed");
138:        super._beforeTokenTransfer(from, to, tokenId, batchSize);
139:    }
```

Description

jayphbee : The comment of `_beforeTokenTransfer` says:
//Only Minter are able to burn/transfer

The require statement only allow `from == address(0)`, that is to say only fresh mint can successed. Nobody can transfer the NFT even if the minter.

```
require(from == address(0), "Tranfer/Burn are not allowed");
```

Further more `ClaimerNFT` contract doesn't implement the `burn` function, so the minter cannot burn NFT.

Recommendation

jayphbee :

```
137: require(hasRole(MINTER_ROLE, msg.sender), "Tranfer/Burn are not allowed");
```

And implment the `burn` function.

```
function burn(uint256 tokenId) public {
    _burn(tokenId);
}
```

Client Response

Declined

Claimer NFT is soulbound token, therefore no transfer/burn. Have changed the misunderstanding comment.

This will not lead to high risk since this is just a misunderstanding.

KTD-8:off-by-one error when validate validatorMaxSupply

Category	Severity	Code Reference	Status	Contributor
Logical	Low	- code/contracts/Validator.sol#L77	Fixed	jayphbee

Code

```
77:        require(tokenIdCounter.current() + tier1Batch + tier2Batch < validatorMaxSupply, "exceed Max Total Supply");
```

Description

jayphbee : There's an edge case that it can only mint `validatorMaxSupply-1` NFTs because off-by-one error. Say calling `initialize` with `initialzie("", "", "", bytes32(0), 5000, 5000)`. This will revert due to

```
require(tokenIdCounter.current() + tier1Batch + tier2Batch < validatorMaxSupply, "exceed Max Total Supply");
```

Recommendation

jayphbee :

```
require(tokenIdCounter.current() + tier1Batch + tier2Batch <= validatorMaxSupply, "exceed Max Total Supply");
```

Client Response

Fixed

KTD-9:omit the roles length check

Category	Severity	Code Reference	Status	Contributor
Logical	Low	• code/contracts/Claimer.sol#L99	Fixed	jayphbee, Hacker007

Code

```
99:        require(to_s.length == validatorTokenIds.length && to_s.length == karatScores.length && to_s.length == lieutenantAddrs.length, "Input arrays must have the same length");
```

Description

jayphbee : In the `mintClaimerBatch` function there are `to_s`, `validatorTokenIds`, `karatScores` and `lieutenantAddrs` length check, but omit the `roles` length check.
Hacker007 : The function `mintClaimerBatch()` does not check the length of the input array `roles` against the length of `to_s`, which may lead to a potential index-out-of-bound error.

Recommendation

jayphbee : Add `roles` length check

```
require(to_s.length == validatorTokenIds.length && to_s.length == karatScores.length && to_s.length == lieutenantAddrs.length && to_s.length == roles.length, "Input arrays must have the same length");
```

Hacker007 : Check if `roles.length` is equal to `to_s.length` as below:

```
require(to_s.length == validatorTokenIds.length && to_s.length == karatScores.length && to_s.length == lieutenantAddrs.length && to_s.length == roles.length, "Input arrays must have the same length");
```

Client Response

Fixed

KTD-10:restrict `msg.value` equal to the exact mint price

Category	Severity	Code Reference	Status	Contributor
Logical	Medium	• code/contracts/Validator.sol#L163 • code/contracts/Validator.sol#L172 • code/contracts/Validator.sol#L180	Fixed	jayphbee, Hacker007

Code

```
163:        require(msg.value >= priceToPay, "Amount Not Correct");

172:        require(msg.value >= price[tier], "Amount Not Correct");

180:        require(msg.value >= price[tier] * amount, "Amount Not Correct");
```

Description

jayphbee : Contract should prevent user from overpaying ether when minting NFT, otherwise user could unexpectedly lose funds.

Hacker007 : In the contract `ValidatorNFT`, the functions `mintValidatorPrivate()`, `mintValidatorPublic`, `mintValidatorPublicBatch` check if the receiving native token is equal to or greater than the expected price.

```
require(msg.value >= priceToPay, "Amount Not Correct");
//...
require(msg.value >= price[tier], "Amount Not Correct");
//...
require(msg.value >= price[tier] * amount, "Amount Not Correct");
```

The problem is that if a user sends a value over the expected price, the excessive tokens won't be refunded, which makes the user lost their tokens and the admin needs to refund his token manually.

Recommendation

jayphbee : restrict msg.value equal to the exact mint price.

```
163: require(msg.value == priceToPay, "Amount Not Correct");
172: require(msg.value == price[tier], "Amount Not Correct");
180: require(msg.value == price[tier] * amount, "Amount Not Correct");
```

Hacker007 : Revisit the validation logic and check if msg.value equals the expected price strictly as below:

```
require(msg.value == priceToPay, "Amount Not Correct");
//...
require(msg.value == price[tier], "Amount Not Correct");
//...
require(msg.value == price[tier] * amount, "Amount Not Correct");
```

Client Response

Fixed

KTD-11:signature replay attack vulnerability in `mintClaimerwithSig` function

Category	Severity	Code Reference	Status	Contributor
Logical	**Critical**	• **code/contracts/Claimer.sol#L110-L124**	**Fixed**	**jayphbee, LiRiu**

Code

```
110:    function mintClaimerwithSig(address to, uint256 validatorTokenId, uint256 karatScore, address lieutenantAddr, Role role, bytes memory signature) public nonReentrant whenNotPaused{
111:        require(balanceOf(to) < 1, "Already Have Token");
112:        bytes32 newHashedMessage = keccak256(
113:                        abi.encode(
114:                            to,
115:                            keccak256(abi.encode(karatScore)),
116:                            keccak256(abi.encode(role))
117:                        )
118:                    );
119:        address signer = recoverSigner(newHashedMessage, signature);
120:
121:        require(hasRole(MINTER_ROLE, signer), "Not Authorized Signer");
122:
123:        _mintClaimer(to, validatorTokenId, karatScore, lieutenantAddr, role);
124:    }
```

Description

jayphbee : The current implementation of `mintClaimerwithSig` might be vulnerable to signature replay attacks. If the `to` address transfer the NFT to other address, he can mint a new NFT again using the same signature previously provided. A nonce system and signature deadline should be added to mitigate this risk.
LiRiu : In the `Claimer.sol::mintClaimerwithSig` function, the user submits `minter.signature` to prove their eligibility for receiving the NFT.

```
    function mintClaimerwithSig(address to, uint256 validatorTokenId, uint256 karatScore, address lieutenantAddr, Role role, bytes memory signature) public nonReentrant whenNotPaused{
        require(balanceOf(to) < 1, "Already Have Token");
        bytes32 newHashedMessage = keccak256(
                        abi.encode(
                            to,
                            keccak256(abi.encode(karatScore)),
                            keccak256(abi.encode(role))
                        )
                    );
        address signer = recoverSigner(newHashedMessage, signature);

        require(hasRole(MINTER_ROLE, signer), "Not Authorized Signer");

        _mintClaimer(to, validatorTokenId, karatScore, lieutenantAddr, role);
    }
```

It is evident that the `newHashedMessage` used for signing consists of "to", "hash(score)", and "hash(role)". The structure of `newHashedMessage` does not comply with EIP-191, and the signer cannot distinguish it from `preSignedTransactionHash`. Attackers can deceive the signer into signing an ETH transaction. This can lead to financial losses.

Click the link to see more discussions about this topic.

PS. Here's a false positive testcase in test/Claimer.ts:L189: `Test mintClaimerwithSig function with valid inputs and a valid signature`. The developer commented out the non-functional signature code to pass the testcase. This issue is precisely the reason why the signature portion of the code became non-functional: `ethers.signMessage` comply with EIP-191, but the contract is not.

LiRiu : Although the `Claimer` contract prevents `mintClaimerwithSig` signature replay attack by checking `balanceOf(msg.sender) < 1`, it does not prevent the risk of cross-contract signature replay or cross-chain signature replay.

```
    function mintClaimerwithSig(address to, uint256 validatorTokenId, uint256 karatScore, address lieutenantAddr, Role role, bytes memory signature) public nonReentrant whenNotPaused{
        require(balanceOf(to) < 1, "Already Have Token");
        bytes32 newHashedMessage = keccak256(
                        abi.encode(
                            to,
                            keccak256(abi.encode(karatScore)),
                            keccak256(abi.encode(role))
                        )
                    );
        address signer = recoverSigner(newHashedMessage, signature);

        require(hasRole(MINTER_ROLE, signer), "Not Authorized Signer");

        _mintClaimer(to, validatorTokenId, karatScore, lieutenantAddr, role);
    }
```

Once the project is deployed on any other chain or re-deployed on the current chain, attackers can replay the "sig" from the previous contract on the new contract to illegally mintNFT on the new contract.

for cross-chain replay: Attacker should search the mintClaimerwithSig transaction that sent from `0x5c82eb01153e4173dc889ec3fb6df8694427b8c9` on those chains:

```
zkSyncTestnet: https://testnet.era.zksync.dev
bsc_testnet: https://data-seed-prebsc-1-s1.binance.org:8545
mumbai: https://polygon-mumbai.g.alchemy.com
goerli: https://eth-goerli.g.alchemy.com
```

for cross-contract replay: you can deploy this PoC contract with signature `0xf38cf66817c794c3e501b2a948a40692523e2370f7559891f73e3de904e8509f3a30461f7d430fcc905c7f9b6cd8c372cf35f080d914c2bf3fc1775edde8eccb1c`

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.18;
import "@openzeppelin/contracts-upgradeable/utils/cryptography/ECDSAUpgradeable.sol";

contract A {
    enum Role {Scientist, Engineer, Doctor, Security, Artist}

    function recoverSigner(bytes32 message, bytes memory signature) public pure returns (address) {
        return ECDSAUpgradeable.recover(message, signature);
    }

    function mintClaimerwithSig(address to, uint256 score, Role role, bytes memory signature) public
view virtual returns(address){
        bytes32 newMessage = keccak256(
                        abi.encode(
                            to,
                            keccak256(abi.encode(score)),
                            keccak256(abi.encode(role))
                        )
                    );
        bytes32 newHashedMessage = ECDSAUpgradeable.toEthSignedMessageHash(newMessage);
        address signer = recoverSigner(newHashedMessage, signature);
        return signer;
    }
}

contract B is A {}

contract PoC {
    A private a;
    B private b;
    bool public pocSuccessful;

    constructor(bytes memory signature) {
        a = new A();
        b = new B();

        address recover_address_a = a.mintClaimerwithSig(0x5B38Da6a701c568545dCfcB03FcB875f56beddC4,
100, A.Role.Doctor, signature);
        address recover_address_b = b.mintClaimerwithSig(0x5B38Da6a701c568545dCfcB03FcB875f56beddC4,
100, A.Role.Doctor, signature);
        require( recover_address_a == recover_address_b, "PoC Failed.");
pocSuccessful = true;
    }
}
```

Recommendation

jayphbee : Add a nonce mapping and include it in the signed message and signture deadline

```
mapping(address => uint256) public nonces;

// ...

function mintClaimerwithSig(address to, uint256 validatorTokenId, uint256 karatScore, address lieutenantAddr, Role role, uint256 nonce, bytes memory signature, uint256 deadline) public nonReentrant whenNotPaused{
    require(balanceOf(to) < 1, "Already Have Token");
    require(nonces[to] == nonce, "Invalid nonce");
    require(deadline >= block.timestamp, "signature expired.");
    nonces[to] += 1;

    bytes32 newHashedMessage = keccak256(
        abi.encode(
            to,
            keccak256(abi.encode(karatScore)),
            keccak256(abi.encode(role)),
            nonce,
            deadline
        )
    );
    address signer = recoverSigner(newHashedMessage, signature);

    require(hasRole(MINTER_ROLE, signer), "Not Authorized Signer");

    _mintClaimer(to, validatorTokenId, karatScore, lieutenantAddr, role);
}
```

Ideally if the `ClaimerNFT` contract intend to deploy to multiple EVM chains, the hashed message should include the chainId.

LiRiu : To standardize the content of a signed message as per EIP-191 just packed newMessage with `ECDSAUpgradeable.toEthSignedMessageHash`

Consider below fix in the `Claimer.mintClaimerwithSig()` function

```
    function mintClaimerwithSig(address to, uint256 validatorTokenId, uint256 karatScore, address lieutenantAddr, Role role, bytes memory signature) public nonReentrant whenNotPaused{
        require(balanceOf(to) < 1, "Already Have Token");
        bytes memory newMessage = abi.encode(
                            to,
                            keccak256(abi.encode(karatScore)),
                            keccak256(abi.encode(role))
                        );
        bytes32 newHashedMessage = ECDSAUpgradeable.toEthSignedMessageHash(newMessage);
        address signer = recoverSigner(newHashedMessage, signature);

        require(hasRole(MINTER_ROLE, signer), "Not Authorized Signer");

        _mintClaimer(to, validatorTokenId, karatScore, lieutenantAddr, role);
    }
```

LiRiu : Adding `contract.address` and `block.chainId` to the signature content can prevent replay attacks across different contracts and chains.

Client Response

Fixed

Since we don’t have a plan to cross the chain, the risk is pretty limited to a small scale. But we still add block chain ID to increase the safety.

KTD-12:use `abi.encode` instead of `abi.encodePacked`

Category	Severity	Code Reference	Status	Contributor
Gas Optimization	Informational	• code/contracts/Claimer.sol#L110-L124	Acknowledged	LiRiu

Code

```
110:    function mintClaimerwithSig(address to, uint256 validatorTokenId, uint256 karatScore, address lieutenantAddr, Role role, bytes memory signature) public nonReentrant whenNotPaused{
111:        require(balanceOf(to) < 1, "Already Have Token");
112:        bytes32 newHashedMessage = keccak256(
113:                        abi.encode(
114:                            to,
115:                            keccak256(abi.encode(karatScore)),
116:                            keccak256(abi.encode(role))
117:                        )
118:                    );
119:        address signer = recoverSigner(newHashedMessage, signature);
120:
121:        require(hasRole(MINTER_ROLE, signer), "Not Authorized Signer");
122:
123:        _mintClaimer(to, validatorTokenId, karatScore, lieutenantAddr, role);
124:    }
```

Description

LiRiu : Since `mintClaimerwithSig` is not available in testcase, the developer will look for the reason why the signature result is different. Most of the answers revolve around the difference between `abi.encode` and `abi.encodePacked`, and recommend using the `abi.encodePakced` function that takes up less space and is more refined.

But this is not the crux of the problem. The correct signature method is in the issue: `Signature Forgery of mintClaimerwithSig function` #2

This issue will discuss two differences between `encode` and `encodePacked`. And explain why I prefer to use the encode function

About Code Readability

for `abi.encode` style contract, we need invoke it in typescript like this:

```
const messageScore = ethers.utils.defaultAbiCoder.encode(
    ["uint256"], [100]
)
const hashScore = ethers.utils.keccak256(messageScore);
const messageRole = ethers.utils.defaultAbiCoder.encode(
    ["uint8"], [2]
)
const hashRole = ethers.utils.keccak256(messageRole);
const messageAll = ethers.utils.defaultAbiCoder.encode(
    ["address", "bytes32", "bytes32"],
    [users[1], hashScore, hashRole]
)
const hashAll = ethers.utils.keccak256(messageAll);
const signature = await owner.signMessage(ethers.utils.arrayify(hashAll));
```

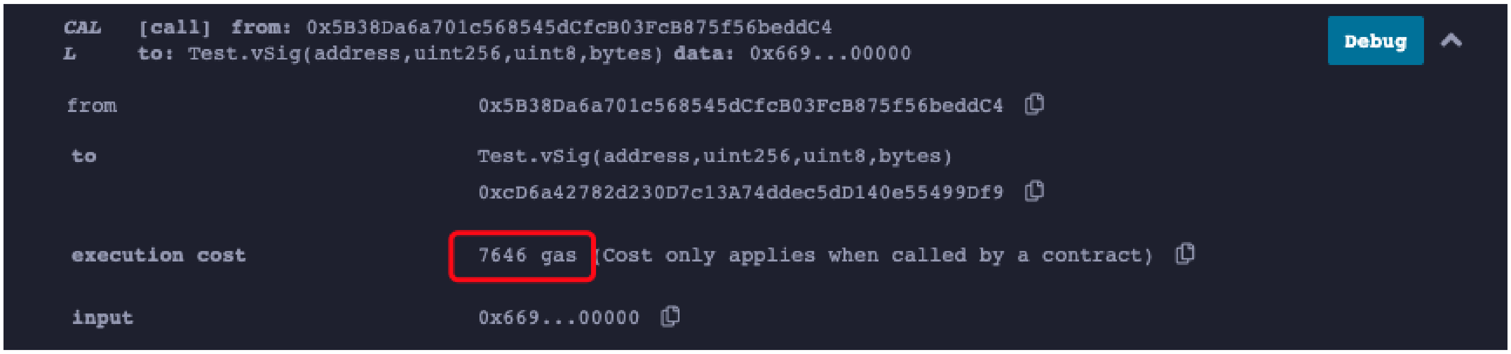
for `abi.encodePacked` style contract, we need invoke it in typescript like this:

```
const message = ethers.utils.solidityKeccak256(
        ["address", "bytes32", "bytes32"],
        [users[1], ethers.utils.solidityKeccak256(["uint256"], [100]), ethers.utils.solidityKeccak25
6(["uint8"], [2])]
      );
const signature = await owner.signMessage(ethers.utils.arrayify(message));
```

For developers, the `abi.encodePacked` style is more refined and elegant, and the code is more readable

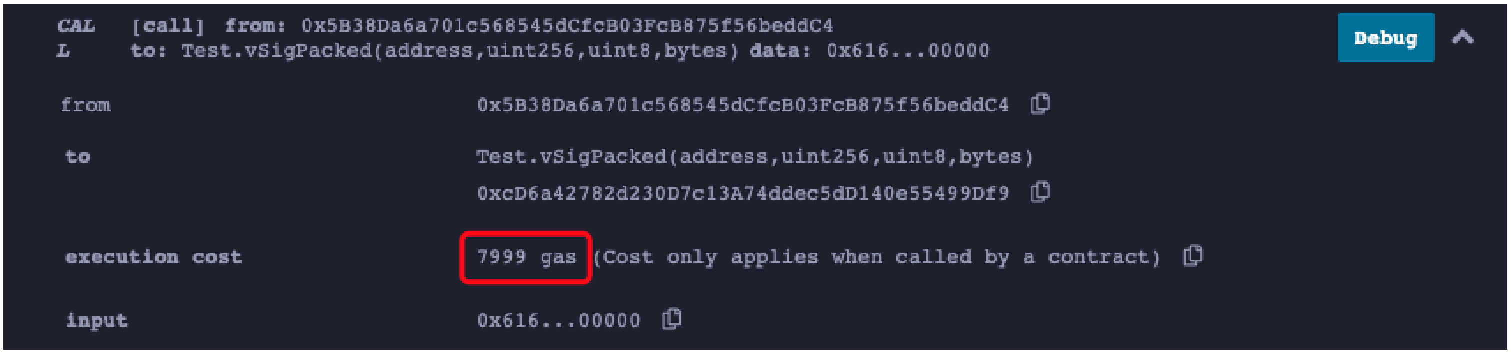
About Gas Cost

for `abi.encode` style contract, gas cost of `mintClaimerwithSig` as below:



```
CAL   [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
L     to: Test.vSig(address,uint256,uint8,bytes) data: 0x669...00000

from              0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

to                Test.vSig(address,uint256,uint8,bytes)
                  0xcD6a42782d230D7c13A74ddec5dD140e55499Df9

execution cost    7646 gas (Cost only applies when called by a contract)

input             0x669...00000
```

for `abi.encodePacked` style contract, gas cost of `mintClaimerwithSig` as below:



```
CAL   [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
L     to: Test.vSigPacked(address,uint256,uint8,bytes) data: 0x616...00000

from              0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

to                Test.vSigPacked(address,uint256,uint8,bytes)
                  0xcD6a42782d230D7c13A74ddec5dD140e55499Df9

execution cost    7999 gas (Cost only applies when called by a contract)

input             0x616...00000
```

For Users, the gas cost of `abi.encodePacked` is higher.

Although encodePacked consumes less calldata space, it uses more opcode. This leads to higher gas consumption

for `encodePacked`: more Readable, but more Gas cost. for `encode`: cheaper

cheaper is better for user.

Recommendation

LiRiu : Stick to abi.encode instead of abi.encodePacked in `mintClaimerwithSig` function.

Client Response

Acknowledged

Will Implement in the future

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3’s prior written consent in each instance.

This report is not an “endorsement” or “disapproval” of any particular project or team. This report is not an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3’s position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.