#HashEx
BLOCKCHAIN SECURITY

NEVUS Medical Service

smart contracts
final audit report

February 2024

hashex.org

contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the Data Ammo team to perform an audit of their smart contract. The audit was conducted between 09/01/2024 and 11/01/2024.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available in the @Data-Ammo/dataammo-contracts GitHub repository after the commit 97713e8.

Update. The Data Ammo team has responded to this report. The updated code is located in the @Data-Ammo/da-con-sectyadt repository and was checked after the commit 36eddd4.

2.1 Summary

Project name	NEVUS Medical Service
URL	https://dataammo.io
Platform	Avalanche Network
Language	Solidity
Centralization level	🔴 High
Centralization risk	🔴 High

2.2 Contracts

Name	Address
MedicalService	

3. Project centralization risks

The project contract is Ownable, i.e., it has restricted functions and a single privileged account to access them. Users using the project have to trust that owner account is properly secured.

C89CR0d Any address can be added as stable coin

The project owner can edit list of supported stable coins, meaning any invoice can be paid without actual payment with owner's support.

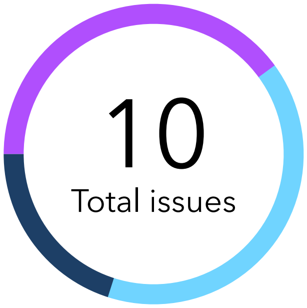
Owner can set any fee percent up to 99.99%, effectively seizing any payment if fee updating transaction are pushed before payment transaction.

Recommendations

Recommendation

We recommend securing the owner's account with use of MultiSig and Timelock contracts.

4. Found issues



●	Medium	4 (40%)
●	Low	4 (40%)
●	Info	2 (20%)

C89. MedicalService

ID	Severity	Title	Status
C89l8a	● Medium	Payment token prices are fixed	✓ Resolved
C89l8b	● Medium	Editing invoce deletes information of amount paid	✓ Resolved
C89l89	● Medium	Possible block gas limit problem	✓ Resolved
C89laf	● Medium	Invoice creation may be frontrun	✓ Resolved
C89l85	● Low	Initialization problem	✓ Resolved
C89l86	● Low	Inconsistent error message	✓ Resolved
C89l8c	● Low	Inconsistent documentation	✓ Resolved
C89l84	● Low	Gas optimizations	✓ Resolved
C89l88	● Info	Same invoice ID can be used by different providers	✓ Acknowledged
C89l87	● Info	Incomplete documentation	✓ Acknowledged

5. Contracts

C89. MedicalService

Overview

The contract allows owner selected providers to create and edit invoices for customers, who can pay those invoices by any of the supported stable coins.

Issues

C89I8a Payment token prices are fixed ● Medium ✓ Resolved

All supported stablecoins are pegged to USD without external confirmation from an oracle. If any of payment tokens become de-pegged, all possible losses will be applied to providers with active invoices. The only possible mitigation is for the owner to recall token support.

Recommendation

Implement oracle price feeds or introduce community governance function to pause specific pament tokens.

C89I8b Editing invoce deletes information of amount paid ● Medium ✓ Resolved

The `editInvoice()` function in the smart contract allows the owner or the provider to edit an existing invoice, including modifying the invoice's payment-related information. The function overrides the paid amount and deletes information related to payment if an invoice has already been paid. This behavior could lead to inconsistencies in the record-keeping of payments and compromise the integrity of the invoicing system.

```
    function editInvoice(
        uint256 _invoiceId,
        uint256 _amount,
        uint256 _startDate,
        uint256 _endDate,
        address _customer,
        uint256 _providerId
    ) external onlyOwnerOrProvider {
        require(invoiceExists[_invoiceId][_providerId], 'Invoice does not exist');
        require(isEnabled[_providerId], 'This provider is disabled');
        require(_amount != 0, 'Payment amount can not be 0');
        require(_startDate < _endDate, 'Wrong date provided');

        uint256 id;
        bool isFound = false;
        for (id = 0; id < invoices[_customer].length; id++) {
            if (invoices[_customer][id].invoiceId == _invoiceId) {
                isFound = true;
                break;
            }
        }
        require(isFound, 'Invoice was not found');

        invoices[_customer][id] = InvoiceInfo({
            paid: false,
            amountPaid: 0,
            token: address(0),
            feeGathered: 0,
            providerId: _providerId,
            invoiceId: _invoiceId,
            amount: _amount,
            startDate: _startDate,
            endDate: _endDate,
            customer: _customer
        });

        emit InvoiceEdited(_invoiceId, _amount, _startDate, _endDate, _providerId,
_customer);
    }
```

Recommendation

Do not update the payment information in the `editInvoice()` function.

C89I89 Possible block gas limit problem ● Medium ✓ Resolved

Possible contract stalling due to block gas limit exceedance in the `payInvoice()`, `getInvoiceInfo()`, `removeInactiveInvoices()`, `removePaidInvoices()`, `removeArchivedInvoices()`, `getInactiveInvoiceAmount()`, `getPaidInvoiceAmount()`, `getArchivedInvoiceAmount()` functions. Too large `invoices[_customer]` array can lead to a case of user being unable to pay, and the owner being unable to reduce the array's length.

```
    function payInvoice(...) external {
        (, InvoiceStatus _status, uint256 index) = getInvoiceInfo(_invoiceId,
_providerId);
        ...
    }

    function getInvoiceInfo(
        uint256 _invoiceId,
        uint256 _providerId
    ) public view returns (InvoiceInfo memory, InvoiceStatus, uint256) {
        address customer = customers[_providerId][_invoiceId];
        for (uint256 i = 0; i < invoices[customer].length; i++) {
            if (
                invoices[customer][i].invoiceId == _invoiceId &&
                invoices[customer][i].providerId == _providerId
            ) {
                return (invoices[customer][i], _getInvoiceStatus(invoices[customer][i]),
i);
            }
        }
        revert('No data about invoice or wrong sender');
    }

    function removeArchivedInvoices(address _customer) external onlyOwner {
        uint256 size = getArchivedInvoiceAmount(_customer);
        ...
        for (uint256 i = 0; i < invoices[_customer].length; i++) {
            if (_isArchivedInvoice(invoices[_customer][i])) indexesToReassemble[index++] =
i;
        }
        _removeInvoiceMany(indexesToReassemble, _customer);
    }

    function getActiveInvoiceAmount(address _customer) public view returns (uint256 count)
{
        for (uint256 i = 0; i < invoices[_customer].length; i++) {
            if (_isActiveInvoice(invoices[_customer][i])) count++;
        }
    }
```

Recommendation

Optimize array search by using EnumerableSet library.

C89Iaf Invoice creation may be frontrun ● Medium ✓ Resolved

Authorized provider can create invoice on behalf of any other provider. Malicious provider can frontrun invoices with a near zero amounts.

```
    /**
     * @dev Creates a new invoice with the given parameters.
     * @param _invoiceId The ID of the invoice.
     * @param _amount The amount of the invoice.
     * @param _startDate The start date of the invoice.
     * @param _endDate The end date of the invoice.
     * @param _customer The address of the customer.
     * @param _providerId ID of the provider.
     */
    function createInvoice(
        uint256 _invoiceId,
        uint256 _amount,
        uint256 _startDate,
        uint256 _endDate,
        address _customer,
        uint256 _providerId
    ) external onlyOwnerOrProvider {
        require(isEnabled[_providerId], 'This provider is disabled');
        require(_amount != 0, 'Payment amount can not be 0');
        require(_invoiceId != 0, 'Invoice id can not be zero');
        require(_startDate < _endDate, 'Wrong date provided');
        require(!invoiceExists[_invoiceId][_providerId], 'This invoice already exists');

        require(
            invoices[_customer].length < maxInvoiceCapacity,
            'Reached maximum invoices for this customer'
        );

        invoices[_customer].push(
            InvoiceInfo({
                paid: false,
                amountPaid: 0,
                token: address(0),
                feeGathered: 0,
                providerId: _providerId,
                invoiceId: _invoiceId,
                amount: _amount,
                startDate: _startDate,
                endDate: _endDate,
                customer: _customer
            })
        );

        invoiceExists[_invoiceId][_providerId] = true;
        customers[_providerId][_invoiceId] = _customer;

        emit InvoiceCreated(_invoiceId, _amount, _startDate, _endDate, _providerId,
_customer);
    }
```

Recommendation

Consider modifying the onlyOwnerOrProvider():

```
    modifier onlyOwnerOrProvider(uint256 _providerId) {
        require((isProvider[msg.sender] &&
                 providers[_providerId].providerAddress == msg.sender) ||
                 owner() == msg.sender, 'Wrong executor');
    _;
}
```

C89I85 Initialization problem

● Low ✓ Resolved

If the `_owner` parameter of the constructor is different from deployer's address, then calls for `updateFee()` and `updateFeeRecipient()` functions during contract deployment will fail due to wrong ownership.

```
constructor(address _owner, uint256 _fee, address _feeRecipient) {
    transferOwnership(_owner);
    updateFee(_fee);
    updateFeeRecipient(_feeRecipient);
}

function updateFee(uint256 _newFee) public onlyOwner { ... }

function updateFeeRecipient(address _newFeeRecipient) public onlyOwner { ... }
```

Recommendation

Move `transferOwnership()` to the last place in the constructor or implement internal `_updateFee()` and `_updateFee()` functions without parameter checks in order to save gas on deployment.

C89I86 Inconsistent error message

● Low ✓ Resolved

Confusing error message L160: provider address is checked but the corresponding error is about provider ID.

```
function addProvider(
    uint256 _providerId,
    address _providerAddress,
    string memory _name,
    string memory _symbol,
    string memory _link
) external onlyOwner {
    ...
    require(!isProvider[_providerAddress], 'This provider id already exists');
    ...
}
```

C89I8c Inconsistent documentation

● Low ✓ Resolved

1. The `editProvider()` function documentation states that the function emits `NewProviderAdded` event instead of `ProviderEdited`.

2. The `getInvoiceStatus()` function documentation states that `InvoiceStatus.Unknown` may be returned. However, there is no such status.

C89I84 Gas optimizations

● Low ✓ Resolved

1. `InvoiceInfo` structure can be repacked to reduce its size: bool + address fields can be placed adjacent to each other.

2. Excessive stored data: the `ProviderInfo` structure contains `providerId` field, which is also stored as a key `providers[providerId].providerId`.

3. The `updateFee()` and `updateFeeRecipient()` functions can have internal logic functions without authorization checks to reduce gas costs for construction.

4. Unnecessary code: L194 requirement condition is always passed as `providerId === providers[providerId].providerId`.

5. Double reads from storage in the `enable()` and `disable()` functions: `isEnabled[_providerId]` is read in the `ProviderStatusUpdated` event.

6. The `removeStableToken()` function can be gas optimized: searching across `stablecoins[]` can be optimized by using an enumerable set, otherwise `stablecoins.length` is read multiple times in `for` loop.

7. The `editInvoice()` function can be gas optimized: searching across `invoices[]` can be optimized by using an enumerable set, otherwise `invoices[_customer].length` is read multiple times in `for` loop.

8. In the `removeInactiveInvoices()`, `removePaidInvoices()`, `removeArchivedInvoices()`, `getInvoiceInfo()`, `getInvoiceStatus()`, `getActiveInvoices()`, `getPaidInvoices()`, `getArchivedInvoices()`, `getInactiveInvoices()`, `getActiveInvoiceAmount()`, `getInactiveInvoiceAmount()`, `getPaidInvoiceAmount()`, `getArchivedInvoiceAmount()` functions `invoices[_customer].length` is read multiple times in `for` loop.

9. Double read of user's `invoices[]` array in the `removeInactiveInvoices()` function: first read is in the `getInactiveInvoiceAmount()` call, second is in the `for` loop.

10. Double read of user's `invoices[]` array in the `removePaidInvoices()` function: first read is in the `getPaidInvoiceAmount()` call, second is in the `for` loop.

11.

Double read of user's `invoices[]` array in the `removeArchivedInvoices()` function: first read is in the `getArchivedInvoiceAmount()` call, second is in the `for` loop.

C89I88 Same invoice ID can be used by different providers ● Info ✓ Acknowledged

Invoice ID is not unique, it can be duplicated by separate providers. This may confuse users and can be potentially used by a malicious actor to fake a valid invoice.

```
/**
 * @dev Creates a new invoice with the given parameters.
 * @param _invoiceId The ID of the invoice.
 * @param _amount The amount of the invoice.
 * @param _startDate The start date of the invoice.
```

```
       * @param _endDate The end date of the invoice.
       * @param _customer The address of the customer.
       * @param _providerId ID of the provider.
       */
      function createInvoice(
          uint256 _invoiceId,
          uint256 _amount,
          uint256 _startDate,
          uint256 _endDate,
          address _customer,
          uint256 _providerId
      ) external onlyOwnerOrProvider {
          ...
          require(_invoiceId != 0, 'Invoice id can not be zero');
          invoiceExists[_invoiceId][_providerId] = true;
          ...
     }
```

C89I87 Incomplete documentation ● Info ✓ Acknowledged

The `createInvoice()` and `editInvoice()` function's parameter `_amount` description should be enriched with mention its decimal precision, which is 18.

```
      /**
       * @dev Creates a new invoice with the given parameters.
       * @param _invoiceId The ID of the invoice.
       * @param _amount The amount of the invoice.
       * @param _startDate The start date of the invoice.
       * @param _endDate The end date of the invoice.
       * @param _customer The address of the customer.
       * @param _providerId ID of the provider.
       */
      function createInvoice(
          uint256 _invoiceId,
          uint256 _amount,
          uint256 _startDate,
          uint256 _endDate,
          address _customer,
          uint256 _providerId
```

```
    ) external onlyOwnerOrProvider { ... }

    /**
     * @dev Calculates the stablecoin value of a given payment amount of an Invoice.
     * @param _token address
     * @param _amount amount of requested payment by provider with 18 decimals value
     * @return value amount of needed payment from customer based on `_token` decimals
     */
    function valueInStablecoin(
        address _token,
        uint256 _amount
    ) public view returns (uint256 value) {
        value = (_amount * 10 ** IERC20Metadata(_token).decimals()) / 10 ** 18;
    }
```

6. Conclusion

4 medium, 4 low severity issues were found during the audit. 4 medium, 4 low issues were resolved in the update. The reviewed contracts are highly dependent on the owner's account. See the centralization risks chapter.

This audit includes recommendations on code improvement and the prevention of potential attacks.

Appendix A. Issues’ severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. Issue status description

✓ **Resolved.** The issue has been completely fixed.

✓ **Partially fixed.** Parts of the issue have been fixed but the issue is not completely resolved.

✓ **Acknowledged.** The team has been notified of the issue, no action has been taken.

? **Open.** The issue remains unresolved.

Appendix C. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

Appendix D. Centralization risks classification

Centralization level

- **High.** The project owners can manipulate user's funds, lock user's funds on their will (reversible or irreversible), or maliciously update contracts parameters or bytecode.
- **Medium.** The project owners can modify contract's parameters to break some functions of the project contract or contracts, but user's funds remain withdrawable.
- **Low.** The contract is trustless or its governance functions are safe against a malicious owner.

Centralization risk

- **High.** Lost ownership over the project contract or contracts may result in user's losses. Contract's ownership belongs to EOA or EOAs, and their security model is unknown or out of scope.
- **Medium.** Contract's ownership is transferred to a contract with not industry-accepted parameters, or to a contract without an audit. Also includes EOA with a documented security model, which is out of scope.
- **Low.** Contract's ownership is transferred to a well-known or audited contract with industry-accepted parameters.

✉ contact@hashex.org

@hashex_manager

blog.hashex.org

linkedin

github

twitter

HashEx

BLOCKCHAIN SECURITY