BlockSec

# Security Audit Report for Unipass wallet contract

**Date:** Feb 1, 2023

**Version:** 2.1

**Contact**: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.02.01 09:55:10 +08'00'

# Contents

## Report Manifest

| Item | Description |
|---|---|
| Client | Unipass |
| Target | Unipass wallet contract |

## Version History

| Version | Date | Description |
|---|---|---|
| 1.0 | Sep 29, 2022 | First Release |
| 1.1 | Oct 17, 2022 | Add `Version 4` |
| 2.0 | Nov 15, 2022 | Second Release (Support OpenID Connect) |
| 2.1 | Feb 1, 2023 | Add `Version 8` |

**About BlockSec** The BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

# Chapter 1 Introduction

## 1.1 About Target Contracts

| Information | Description |
|---|---|
| Type | Smart Contract |
| Language | Solidity |
| Approach | Semi-automatic and manual verification |

The repository that has been audited includes Unipass-Wallet-Contract [1]. The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following.

| Project | | Commit SHA |
|---|---|---|
| Unipass-Wallet-Contract | Version 1 | 31b633a71a5853bf90e9f3ed290414063e5abc53 |
| | Version 2 | fdf34f529e136190df97dbbf8812e51cff50a85c |
| | Version 3 | 68936b4c512a1a133b0e0d890b07038ab6269f88 |
| | Version 4 | 1a0f0e333c24d174d7dd057621b440be6bc51202 |
| | Version 5 | 9d29950ca183d9f26af405506ddccbfe78d92f1d |
| | Version 6 | 43a8fb3d89ecf0b5091d3e3b4897b4cb222b51a2 |
| | Version 7 | 5dfff1cf3ee1095cd7213e4b3b0a52b4b3154304 |
| | Version 8 | b5de524eabc036522a2f47349b88836bd7376c5c |

Note that, we did **NOT** audit smart contracts for testing, including ModuleIgnoreAccount, ModuleIgnore-AuthUpgradable, ModuleMainGasEstimator, and contracts in "tests" folder.

## 1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

[1] https://github.com/UniPassID/Unipass-Wallet-Contract.git

## 1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

### 1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

### 1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

### 1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

### 1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

**Note** *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

## 1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

**Table 1.1:** Vulnerability Severity Classification

| Impact \ Likelihood | High | Low |
| --- | --- | --- |
| *High* | High | Medium |
| *Low* | Medium | Low |

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

# Chapter 2 Findings

In total, we find **four** potential issues. We have **four** recommendations.

- High Risk: 2
- Medium Risk: 1
- Low Risk: 1
- Recommendations: 4
- Notes: 3

| ID | Severity | Description | Category | Status |
|---|---|---|---|---|
| 1 | High | The illegal escalation of privileges I | DeFi Security | Fixed |
| 2 | High | The illegal escalation of privileges II | DeFi Security | Fixed |
| 3 | Medium | The lack of access control | DeFi Security | Fixed |
| 4 | Low | The lack of an external function to update DkimZK | DeFi Security | Fixed |
| 5 | - | Remove the unused functions | Recommendation | Acknowledged |
| 6 | - | Fix the dead code | Recommendation | Fixed |
| 7 | - | Make the comments and code consistent | Recommendation | Fixed |
| 8 | - | Fix typos | Recommendation | Fixed |
| 9 | - | The external call of IERC1271 wallet | Notes | Acknowledged |
| 10 | - | The discussion about the validation of Dkim signature | Notes | Confirmed |
| 11 | - | The discussion about the validation of ID Token | Notes | Confirmed |

## 2.1 DeFi Security

### 2.1.1 The illegal escalation of privileges I

**Severity** High

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** As shown in the below code, Unipass wallet validates owner weight, assets' operation weight, and guardian weight if the transaction is self-invoking. Otherwise, it validates only assets' operation weight. However, the transaction's call type can be delegate-call. Note that, all state changes through the delegate call are applied to the caller contract. Therefore, a delegate call transaction will cause the illegal escalation of privileges.

```
if (transaction.target == address(this)) {
  (uint32 ownerWeight, uint32 assetsOpWeight, uint32 guardianWeight) = _getPermissionOfCallData(transaction.data);

  require(
      _ownerWeight >= ownerWeight && _assetsOpWeight >= assetsOpWeight && _guardianWeight >= guardianWeight,
      "_execute: INVALID_ROLE_WEIGHT"
```

```
    );
} else {
    require(_assetsOpWeight >= LibRole.ASSETS_OP_THRESHOLD, "_executeOnce: INVALID_ROLE_WEIGHT")
        ;
}

if (transaction.callType == CallType.Call) {
    success = LibOptim.call(
        transaction.target,
        transaction.value,
        gasLimit == 0 ? gasleft() : gasLimit,
        transaction.data
    );
} else if (transaction.callType == CallType.DelegateCall) {
    success = LibOptim.delegatecall(transaction.target, gasLimit == 0 ? gasleft() : gasLimit,
        transaction.data);
}
```

**Listing 2.1:** ModuleCall.sol

**Impact** A delegate call transaction will cause the illegal escalation of privileges: from the assets' operation privilege to the owner and guardian privilege.

**Suggestion** Forbid the use of the delegate call in the function `_execute`.

### 2.1.2 The illegal escalation of privileges II

**Severity** High

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** As shown in the below code, if the function selector is `selfExecute.selector`, the return values `ownerWeight`, `assetsWeight`, and `guardianWeight` will be overwritten to zero by the code in line 155. As a result, the permission check of functions `addHook`, `removeHook`, `addPermission`, and `removePermission` can be bypassed by invoking the function `selfExecute`. That's to say any user (without any privilege) can invoke `addHook`, `removeHook`, `addPermission`, and `removePermission`, which originally require the owner privilege.

```
135   function _getPermissionOfCallData(bytes calldata callData)
136       private
137       view
138       returns (
139           uint32 ownerWeight,
140           uint32 assetsWeight,
141           uint32 guardianWeight
142       )
143   {
144       uint256 index;
145       bytes4 selector;
146       (selector, index) = callData.cReadBytes4(index);
147       if (selector == this.selfExecute.selector) {
148           ownerWeight = uint32(uint256(callData.mcReadBytes32(index)));
```

```
149            index += 32;
150            assetsWeight = uint32(uint256(callData.mcReadBytes32(index)));
151            index += 32;
152            guardianWeight = uint32(uint256(callData.mcReadBytes32(index)));
153            index += 32;
154        }
155        (ownerWeight, assetsWeight, guardianWeight) = getRoleOfPermission(selector);
156    }
```

**Listing 2.2:** ModuleCall.sol

**Impact** A transaction invoking `selfExecute` may cause the illegal escalation of privileges: from no privilege to the owner privilege.

**Suggestion** Add a critical word `else` in line $155$.

### 2.1.3 The lack of access control

**Severity** Medium

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The below two functions `updateHookWhiteList` and `updateImplementationWhiteList` should be privileged functions and the contract ModuleWhiteList inherits the contract ModuleAdminAuth. However, the two functions are not decorated by the modifier `onlyAdmin`, which means they can be accessed by anyone.

```
function updateHookWhiteList(address _addr, bool _isWhite) external {
    bool isWhite = hooks[_addr];
    if (isWhite != _isWhite) {
        hooks[_addr] = _isWhite;
        emit UpdateHookWhiteList(_addr, _isWhite);
    } else {
        revert InvalidStatus(isWhite, _isWhite);
    }
}
```

**Listing 2.3:** ModuleWhiteList.sol

```
function updateImplementationWhiteList(address _addr, bool _isWhite) external {
    bool isWhite = implementations[_addr];
    if (isWhite != _isWhite) {
        implementations[_addr] = _isWhite;
        emit UpdateImplementationWhiteList(_addr, _isWhite);
    } else {
        revert InvalidStatus(isWhite, _isWhite);
    }
}
```

**Listing 2.4:** ModuleWhiteList.sol

**Impact** Anyone can update the white list of hook contracts and implementation contracts.

**Suggestion** Add the modifier `onlyAdmin` for the two functions.

### 2.1.4 The lack of an external function to update DkimZK

**Severity** Low

**Status** Fixed in Version 7

**Introduced by** Version 6

**Description** As shown in below code, there is an inaccessible internal function _writeDkimZK, which means DkimKeys can not update the DkimZK contract's address.

```
function _writeDkimZK(IDkimKeys _dkimZK) internal {
    ModuleStorage.writeBytes32(DKIM_ZK_KEY, bytes32(bytes20(address(_dkimZK))));
}
```

**Listing 2.5:** DkimKeys.sol

**Impact** There may have a compatibility issue after the project upgrading the DkimZk contract.

**Suggestion** Add an authorized function to update the DkimZK contract's address.

## 2.2 Additional Recommendation

### 2.2.1 Remove the unused functions

**Status** Acknowledged

**Introduced by** Version 1

**Description** The below three functions are not used.

```
function _parseRoleWeight(uint256 _index, bytes calldata _signature)
    private
    pure
    returns (
        uint32 ownerWeight,
        uint32 assetsOpWeight,
        uint32 guardianWeight,
        uint256 index
    )
{
    (ownerWeight, index) = _signature.cReadUint32(_index);
    (assetsOpWeight, index) = _signature.cReadUint32(index);
    (guardianWeight, index) = _signature.cReadUint32(index);
}
```

**Listing 2.6:** ModuleAuth.sol

```
function checkEmailFrom(bytes calldata _emailFrom, bytes32 _sdid) internal pure returns (bytes
     memory emailFromRet) {
    uint256 atSignIndex = _emailFrom.findBytes1(0, AtSignBytes1);
    bytes32 domain = _emailFrom.mcReadBytes32(atSignIndex + 1);

    require(domain == bytes32("mail.unipass.me") || domain == _sdid, "ED");

    if (
```

```
        _sdid == bytes32("gmail.com") ||
        _sdid == bytes32("googlemail.com") ||
        _sdid == bytes32("protonmail.com") ||
        _sdid == bytes32("proton.me") ||
        _sdid == bytes32("pm.me")
    ) {
        emailFromRet = removeDotForEmailFrom(_emailFrom, atSignIndex);
    } else {
        emailFromRet = _emailFrom;
    }
    emailFromRet = emailFromRet.toLowerMemory();
}
```

Listing 2.7: DkimKeys.sol

```
function removeDotForEmailFrom(bytes calldata _emailFrom, uint256 _atSignIndex) internal pure
    returns (bytes memory fromRet) {
    uint256 leftIndex;
    for (uint256 index; index < _atSignIndex; index++) {
        fromRet = leftIndex == 0 ? _emailFrom[leftIndex:index] : bytes.concat(fromRet,
            _emailFrom[leftIndex:index]);
        leftIndex = index;
    }
    if (leftIndex == 0) {
        fromRet = _emailFrom;
    } else {
        bytes.concat(fromRet, _emailFrom[_atSignIndex:_emailFrom.length]);
    }
}
```

Listing 2.8: DkimKeys.sol

**Impact** NA.

**Suggestion** Remove the unused functions.

### 2.2.2 Fix the dead code

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** As shown in the below code, there is no code assigning a value to the variable `tmpEmailType`. Therefore, the code in line 115 to 119 are dead code.

```
109 while (_index < _signature.length - 1) {
110     IDkimKeys.EmailType tmpEmailType;
111     bool isSig;
112     LibUnipassSig.KeyType keyType;
113     bytes32 ret;
114     (isSig, emailType, keyType, ret, _index) = LibUnipassSig._parseKey(dkimKeys, _hash,
            _signature, _index);
115     if (emailType == IDkimKeys.EmailType.None && tmpEmailType != IDkimKeys.EmailType.None) {
116         emailType = tmpEmailType;
```

```
        } else if (emailType != IDkimKeys.EmailType.None && tmpEmailType != IDkimKeys.EmailType.None) {
            require(emailType == tmpEmailType, "_validateSignatureInner: INVALID_EMAILTYPE");
        }
        uint96 singleWeights = uint96(bytes12(_signature.mcReadBytesN(_index, 12)));
        _index += 12;
        if (isSig) {
            weights += singleWeights;
        }
        if (keyType == LibUnipassSig.KeyType.Secp256k1 || keyType == LibUnipassSig.KeyType.ERC1271Wallet) {
            keysetHash = keysetHash == bytes32(0)
                ? keccak256(abi.encodePacked(keyType, address(uint160(uint256(ret))), singleWeights))
                : keccak256(abi.encodePacked(keysetHash, keyType, address(uint160(uint256(ret))), singleWeights));
        } else {
            keysetHash = keysetHash == bytes32(0)
                ? keccak256(abi.encodePacked(keyType, ret, singleWeights))
                : keccak256(abi.encodePacked(keysetHash, keyType, ret, singleWeights));
        }
    }
```

**Listing 2.9:** ModuleAuth.sol

**Impact** The dead code hinders auditors from understanding developers' intent.

**Suggestion** Fix the dead code.

### 2.2.3 Make the comments and code consistent

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The below two functions `updateHookWhiteList` and `updateImplementationWhiteList` and their comments are inconsistent.

```
    /**
     * @dev For mapping whilteList.whiteList, value is the index of whilteList.addresses + 1.
     *      If value == 0, address not exists, if value > 0, value - 1 equals addresses' index.
     * @param _addr Whilte List Address
     * @param _isWhite Add _addr to white list or remove from white list
     */
    function updateHookWhiteList(address _addr, bool _isWhite) external {
        bool isWhite = hooks[_addr];
        if (isWhite != _isWhite) {
            hooks[_addr] = _isWhite;
            emit UpdateHookWhiteList(_addr, _isWhite);
        } else {
            revert InvalidStatus(isWhite, _isWhite);
        }
    }
```

**Listing 2.10:** ModuleWhiteList.sol

```
/**
* @dev For mapping whilteList.whiteList, value is the index of whilteList.addresses + 1.
*     If value == 0, address not exists, if value > 0, value - 1 equals addresses' index.
* @param _addr Whilte List Address
* @param _isWhite Add _addr to white list or remove from white list
*/
function updateImplementationWhiteList(address _addr, bool _isWhite) external {
    bool isWhite = implementations[_addr];
    if (isWhite != _isWhite) {
        implementations[_addr] = _isWhite;
        emit UpdateImplementationWhiteList(_addr, _isWhite);
    } else {
        revert InvalidStatus(isWhite, _isWhite);
    }
}
```

**Listing 2.11:** ModuleWhiteList.sol

**Impact** NA.

**Suggestion** Make the comments and code consistent.

### 2.2.4 Fix typos

**Status** Fixed in `Version 6`

**Introduced by** `Version 5`

**Description** Here are a few typos:

```
/**
* openIDAudience: keccak256(issuser + audiance) => is valid
*/
```

**Listing 2.12:** OpenID.sol

`audiance` -> `audience`

```
function updateOpenIDPublidKey(bytes32 _key, bytes calldata _publicKey) external onlyAdmin {
```

**Listing 2.13:** OpenID.sol

`updateOpenIDPublidKey` -> `updateOpenIDPublicKey`

```
require(suffix == bytes2('",') || suffix == bytes2('"}'), "_getIss: INVALID_KID_RIGHT");
```

**Listing 2.14:** OpenID.sol

`_getIss` -> `_getKid`

**Impact** NA.

**Suggestion** Fix typos.

## 2.3 Notes

### 2.3.1 The external call of IERC1271 wallet

**Status** Acknowledged

**Introduced by** `Version 1`

**Description** As shown in the below code, there is a signature type named ERC1271Wallet that is expected to verify that the specified signature is signed by the ERC1271Wallet's owner. Note that, in the Unipass contract, the signature validation is completed by an external call `IERC1271(key).isValidSignature(_hash, sig)`. Since the uncertainty of external call, we write the note here to remind users to set the secure and correct ERC1271 wallet address.

```
} else if (keyType == KeyType.ERC1271Wallet) {
    isSig = _signature.mcReadUint8(index) == 1;
    ++index;
    address key;
    (key, index) = _signature.cReadAddress(index);
    if (isSig) {
        uint32 sigLen;
        (sigLen, index) = _signature.cReadUint32(index);
        bytes calldata sig = _signature[index:index + sigLen];
        index += sigLen;
        require(
            IERC1271(key).isValidSignature(_hash, sig) == SELECTOR_ERC1271_BYTES32_BYTES,
            "_validateSignature: VALIDATE_FAILED"
        );
    }
    ret = bytes32(uint256(uint160(key)));
} else if (keyType == KeyType.EmailAddress) {
```

**Listing 2.15:** LibUnipassSig.sol

### 2.3.2 The discussion about the validation of Dkim signature

**Status** Confirmed

**Introduced by** `Version 1`

**Description** Unipass wallet contract allows users to manage wallet with email addresses with Dkim signatures. The contract DkimKeys is responsible to validate the email header:

1. Extract the "from" email account from the header
2. Extract the digest hash from the header's subject field
3. Validate the Dkim signature of the header using public keys of supported providers
4. Ensure the "from" email account is the same with the account stored in the DkimKeys contract

Note that, the RSA signature authentication guarantees that the email was sent from the support email providers, e.g. gmail. However, it can not guarantee that it was sent from a specified email account. In order to verify the email sender address, the contract needs to verify the email sender from extracting the “from" field in the signed email headers.

As mentioned above, the critical part of the validation is the way to extract the "from" field, and is it possible for an potential attacker to forge the “from" field?

```
function _getEmailFrom(
    bytes32 _pepper,
    bytes calldata _data,
    uint256 _index,
    bytes calldata _emailHeader
) internal pure returns (bytes32 emailHash) {
    uint32 fromIndex;
    uint32 fromLeftIndex;
    uint32 fromRightIndex;

    (fromIndex, ) = _data.cReadUint32(_index + uint256(DkimParamsIndex.fromIndex) * 4);
    (fromLeftIndex, ) = _data.cReadUint32(_index + uint256(DkimParamsIndex.fromLeftIndex) * 4);
    (fromRightIndex, ) = _data.cReadUint32(_index + uint256(DkimParamsIndex.fromRightIndex) *
        4);
    if (fromIndex != 0) {
        require(_emailHeader.mcReadBytesN(fromIndex - 2, 7) == bytes32("\r\nfrom:"), "FE");
    } else {
        require(_emailHeader.mcReadBytesN(fromIndex, 5) == bytes32("from:"), "FE");
    }
    // see https://www.rfc-editor.org/rfc/rfc2822#section-3.4.1
    require(fromIndex + 4 < fromLeftIndex && fromLeftIndex < fromRightIndex, "LE");
    if (_emailHeader[fromLeftIndex - 1] == "<" && _emailHeader[fromRightIndex + 1] == ">") {
        for (uint256 i = fromLeftIndex - 1; i > fromIndex + 4; i--) {
            require(_emailHeader[i] != "\n", "NE");
        }
    } else {
        require(fromLeftIndex == fromIndex + 5, "AE");
    }

    emailHash = LibEmailHash.emailAddressHash(_emailHeader[fromLeftIndex:fromRightIndex + 1],
        _pepper);
}
```

**Listing 2.16:** DkimKeys.sol

As shown in above code, it extracts the "from" field from the email header using three **externally specified** cursors: `fromIndex`, `fromLeftIndex`, and `fromRightIndex`. With the crafted cursors, we provide two potential methods to forge the "from" field that can cheat the above function. First, we assume the authorized email account is "authorized@gmail.com".

1. Register an email account: "authorized@gmail.com@gmail.com" and set `fromRightIndex` in front of the second "@gmail.com".
2. Insert the string: "CRLFfrom:authorized@gmail.com" in the subject field, and set the three cursors to point to it.

However, the two methods both are **practically infeasible**. That’s because common email providers do not allow users to register an email account containing the ’@’ char, and they also do not allow insert CRLF into subject usually.

In summary, although we did not find a feasible way to attack the function `_getEmailFrom`, we believe the security of the project depends on specific rules internally enforced by supported email providers, as shown in the following.

1. **Can not** allow users to register an email account containing the '@' character.
2. **Can not** leak the private keys to any potential attackers.
3. **Can not** allow users to modify any fields in email header except for the 'subject' field and 'to' field.
4. **Can not** sign an email header with the subject containing a string that is an arbitrary character following a CRLF.

This may become a security loophole if any email provider does not follow the previous rules. We strongly recommend that the project will check above rules when supporting a new email provider.

**Feedback from the Project** We only support authoritative email service providers that will never leak private keys to any individuals, and they all follow some RFCs. For example,

1. section 3.4.1 of RFC5322 [1] stipulates the account specification that **does not allow** to contain the '@' character.
2. section 3.5 of RFC6376 [2] **does not allow** users to modify fields in an email header except for the 'subject' field and 'to' field.
3. section 2.2 of RFC5322 [3], section 3.4.1 and section 3.4.2 of RFC6376 [4] **do not allow** a CRLF contained in the subject field of an email header.

### 2.3.3 The discussion about the validation of ID Token

**Status** Confirmed

**Introduced by** `Version 5`

**Description** Unipass wallet contract allows users to manage wallet using OpenID Connect (OIDC) that is an identity layer built on top of the OAuth 2.0 framework. [5] The contract OpenID is responsible to validate the ID token granted by the authorization server, e.g. Google, which is as follows:

1. Extract fields: "iss", "aud", "sub", "kid", "iat", "exp", "nonce", and "signature" from the ID token.
   - "iss": the ID token issuer, e.g. "https://accounts.google.com".
   - "kid": the key ID through that the corresponding public key of the authorization server can be retrieved.
   - "signature": the signature on the ID token (except for the "signature" field) with the private key corresponding to "kid".
   - "aud": the client ID that the project applies for the Unipass front-end application on the authorization server.
   - "sub": the UUID that the authorization server generates for its users, which are wallets' owners in this scenario.
   - "iat": the issuance time of the ID token.
   - "exp": the expiration time of the ID token.
   - "nonce": an arbitrary value passed by the client to mitigate replay attack, which is the digest hash of transactions the wallet will execute.

[1] https://datatracker.ietf.org/doc/html/rfc5322#section-3.4.1

[2] https://www.rfc-editor.org/rfc/rfc6376#section-3.5

[3] https://datatracker.ietf.org/doc/html/rfc5322#section-2.2

[4] https://www.rfc-editor.org/rfc/rfc6376#section-3.4.1

[5] https://auth0.com/docs/authenticate/protocols/oauth

2. Validate that the block's timestamp is within the time range specified by "iat" and "exp".
3. Validate that the hash of "iss"-"aud" is preset in the contract's whitelist.
4. Load the public key corresponding to "iss"-"kid" from the contract's storage and validate that the signature is signed by the public key.
5. Validate that "nonce" is the digest hash.
6. Validate that "iss"-"sub" is preset in the contract.

During the generation of ID token, there are few fields of ID token can be manipulated by potential attackers. First, "iss", "kid", "signature", "iat", and "exp" are generated and filled by the authorization server, e.g. Google. Second, "sub" is filled by the wallet owner passing the password verification. Third, "aud" and "nonce" are passed by the client that is supposed to be the Unipass front-end application. Under our threat model, the authorization server is credible but the front-end is not, because the client ID is public and potential attackers can use the same "aud" with the Unipass front-end application.

Therefore, the only security concern is that "nonce" can be manipulated to cheat the OpenID contract. For example, the attacker can append a faked "sub" field at the end of "nonce", and then set the `subLeftIndex` and `subRightIndex` to refer to the faked "sub". As a result, the attacker can use other people's (the faked "sub") wallet. However, we cannot find a feasible way to perform that, because the OpenID contract and Google both have some checks to avoid that.

```
function _getSub(
    uint256 _index,
    bytes calldata _data,
    bytes calldata _payload
) internal pure returns (bytes calldata sub) {
    uint32 subLeftIndex;
    (subLeftIndex, ) = _data.cReadUint32(uint256(OpenIDParamsIndex.subLeftIndex) * 4 + _index);
    require(bytes7(_payload[subLeftIndex - 7:subLeftIndex]) == bytes7('"sub":"'), "_getSub:
        INVALID_SUB_LEFT");

    uint32 subRightIndex;
    (subRightIndex, ) = _data.cReadUint32(uint256(OpenIDParamsIndex.subRightIndex) * 4 + _index
        );
    bytes2 suffix = bytes2(_payload[subRightIndex:subRightIndex + 2]);
    require(suffix == bytes2('",') || suffix == bytes2('"}'), "_getSub: INVALID_SUB_RIGHT");

    sub = _payload[subLeftIndex:subRightIndex];
}
```

**Listing 2.17:** OpenID.sol

As shown as the above code, the OpenID contract extracts the "sub" field by matching a pattern: `"sub:"......`. Therefore, in order to cheat the contract, the attacker should append a string like `"sub:"......` at the end of the "nonce" field. However, Google escapes the double quotes `"` in the "nonce" field to `\"`. Under that, potential attackers cannot insert any legal field prefix in the "nonce" field.

In summary, the OpenID design is safe only with a credible authorization server. We recommend that the project must confirm the "nonce" field injection is infeasible before supporting a new authorization server.

In addition, since the client ID is public, potential attackers can leverage phishing website to trick wallet owners into transferring money to attackers. We also remind users to access the correct Unipass website.