// HALBORN

Haqq - Social

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **August 7th, 2023 - August 11th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	08/09/2023	Piotr Cielas
0.2	Draft Version	08/10/2023	Piotr Cielas
0.3	Draft Review	08/11/2023	Gabi Urrutia
1.0	Remediation Plan	09/02/2023	Piotr Cielas
1.1	Remediation Plan Review	09/05/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Piotr Cielas	Halborn	Piotr.Cielas@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

The Social smart contracts by Haqq supports their implementation of Shamir's Secret Sharing system by storing data required to reconstruct users' secrets.

Haqq engaged Halborn to conduct a security assessment on their smart contracts beginning on August 7th, 2023 and ending on August 11th, 2023. The security assessment was scoped to the smart contracts provided in the social GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

1.2 ASSESSMENT SUMMARY

Halborn was provided 1 week for the engagement and assigned 1 full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security experts with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some security risks that were mostly addressed by Haqq.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard

to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs (MythX).
- Static Analysis of security for scoped contract, and imported functions (Slither).
- Testnet deployment (Foundry).

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

Code repositories:

1. Social

- Repository: social
- Commit IDs:
  - Initial: e12c8de
  - Remediation Plan: 4ef69aa
- Smart contracts in scope:
  1. VerifierList (contracts/contracts/VerifierList.sol)
  2. RelayerList (contracts/contracts/RelayerList.sol)
  3. RelayerStorage (contracts/contracts/RelayerStorage.sol)

Out-of-scope

- Third-party libraries and dependencies.
- Economic attacks.

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	1	2	4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) ANONYMOUS ACCESS TO PRIVILEGED FUNCTIONS	Critical (10)	SOLVED - 08/30/2023
(HAL-02) VERIFIER DATA CANNOT BE FULLY DELETED	Medium (5.0)	SOLVED - 08/30/2023
(HAL-03) ZERO ADDRESS AND ZERO LENGTH CHECKS MISSING	Low (2.5)	SOLVED - 09/04/2023
(HAL-04) REDUNDANT STORAGE DATA	Low (2.5)	SOLVED - 08/30/2023
(HAL-05) SINGLE STEP OWNERSHIP TRANSFER PROCESS	Informational (0.7)	SOLVED - 08/30/2023
(HAL-06) IMMUTABLE RELAYER ADDRESSES ARRAY	Informational (0.7)	SOLVED - 09/04/2023
(HAL-07) REDUNDANT COMPUTATION INSIDE LOOP	Informational (0.0)	SOLVED - 08/30/2023
(HAL-08) INEFFICTIENT FOR LOOPS	Informational (0.0)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

4.1 (HAL-01) ANONYMOUS ACCESS TO PRIVILEGED FUNCTIONS - CRITICAL(10)

Description:

The VerifierList contract stores and manages Verifier records for Haqq Social. Verifier objects persist the data required to verify user JWTs when received from a KeyNode. Only the contract owner is allowed to add new Verifiers however the deleteVerifier function lacks such authorization checks and so it allows everyone to delete Verifiers added by the contract owner.

When exploited by a malicious entity, this vulnerability can effectively halt the entire Social system.

A similar issue was identified in the RelayerList contract, which manages Relayer data. Relayers are used by the mobile app to fetch user identity data. The createRelayerList function does not require any authorization and allows anonymous access to create and updating Relayer lists for arbitrary users. This vulnerability, if exploited, can lead to serious disruptions in the off-chain components of the Social network.

Code Location:

Listing 1: contracts/contracts/VerifierList.sol (Line 74)

```
72 function deleteVerifier(string memory _id) external  {
73     require(keccak256(abi.encodePacked(verifiers[_id].id)) !=
↳ keccak256(abi.encodePacked("")), "Verifier does not exist");
74     delete verifiers[_id];
75 }
```

Listing 2: contracts/contracts/RelayerList.sol

```
55 function createRelayerList(bytes32 socialIDHash) external {
56     address[] memory copiedRelayerAddresses = new address[](
↳ relayerAddresses.length);
```

```
57      for (uint256 i = 0; i < relayerAddresses.length; i++) {
58          copiedRelayerAddresses[i] = relayerAddresses[i];
59      }
60
61      uint256 copiedLength = relayerAddresses.length;
62  }
```

Proof Of Concept:

VerifierList

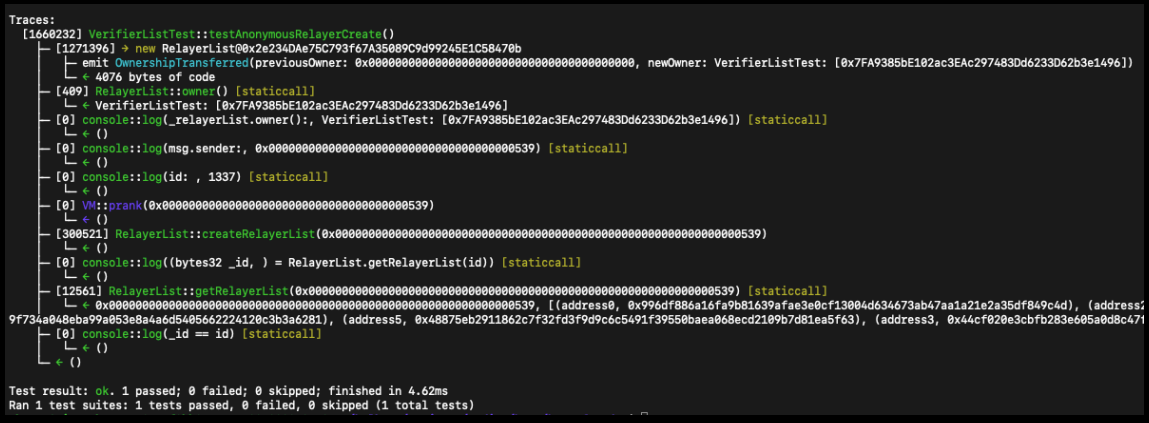
1. Deploy the VerifierList contract.
2. As the contract owner, add a new Verifier.
3. As another user, delete the recently added Verifier.

```
Traces:
  [156188] VerifierListTest::testAnonymousVerifierDelete()
    ├─ [0] console::log(verifier id: , test) [staticcall]
    │   └─ ← ()
    ├─ [2346] VerifierList::owner() [staticcall]
    │   └─ ← VerifierListTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496]
    ├─ [0] console::log(adding a verifier as the owner: , VerifierListTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496]) [staticcall]
    │   └─ ← ()
    ├─ [162595] VerifierList::addVerifier(test, test, test, test, [(test, test)])
    │   └─ ← ()
    ├─ [0] console::log(removing a verifier as a random user: , 0x0000000000000000000000000000000000000539) [staticcall]
    │   └─ ← ()
    ├─ [0] VM::prank(0x0000000000000000000000000000000000000539)
    │   └─ ← ()
    ├─ [2809] VerifierList::deleteVerifier(test)
    │   └─ ← ()
    ├─ [0] console::log(getting the verifier with id: , test) [staticcall]
    │   └─ ← ()
    ├─ [4722] VerifierList::getVerifier(test) [staticcall]
    │   └─ ← , , , , []
    ├─ [0] console::log(fetched verifier id: , ) [staticcall]
    │   └─ ← ()
    └─ ← ()

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.56ms
```

RelayerList

1. Deploy the RelayerList contract.
2. As any user, create a Relayer list for any social ID.



BVSS:

AO:A/AC:L/AX:L/C:N/I:C/A:N/D:N/Y:N/R:N/S:U (10)

Recommendation:

To prevent anonymous users from altering sensitive areas of contract state, consider protecting the deleteVerifier and createRelayerList functions with the onlyOwner modifier.

Remediation Plan:

SOLVED: The Haqq team fixed this issue in commit 9fcd3e1. The deleteVerifier and createRelayerList functions are now protected with the onlyOwner modifier.

4.2 (HAL-02) VERIFIER DATA CANNOT BE FULLY DELETED - MEDIUM (5.0)

Description:

The VerifierList contract stores and manages Verifier records for Haqq Social. Verifier objects persist the data required to verify user JWTs when received from a KeyNode. Only the contract owner is allowed to add new Verifiers. A verifier can be removed from storage with the deleteVerifier function, which uses the delete keyword on a Verifier at a user-supplied id. However, because the Verifier structs store the checks mappings, those mappings cannot be reset with delete which causes the deleteVerifier function to fail silently.

This issue may disrupt the operations of the off-chain components of the Social network.

Code Location:

Listing 3: contracts/contracts/VerifierList.sol (Line 72)

```
70 function deleteVerifier(string memory _id) external  {
71     require(keccak256(abi.encodePacked(verifiers[_id].id)) !=
↳ keccak256(abi.encodePacked("")), "Verifier does not exist");
72     delete verifiers[_id];
73 }
```

Proof Of Concept:

VerifierList

1. Deploy the VerifierList contract.
2. As the contract owner, add a new Verifier.
3. Delete the recently added Verifier.
4. Query the contract for any check for the Verifier.

FINDINGS & TECH DETAILS

```
Traces:
  [162748] VerifierListTest::testAnonymousVerifierDelete()
    ├─ [0] console::log(verifier id: , test) [staticcall]
    │   └─ ← ()
    ├─ [2346] VerifierList::owner() [staticcall]
    │   └─ ← VerifierListTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496]
    ├─ [0] console::log(adding a verifier as the owner: , VerifierListTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496]) [staticcall]
    │   └─ ← ()
    ├─ [162595] VerifierList::addVerifier(test, test, test, test, [(test, test)])
    │   └─ ← ()
    ├─ [0] console::log(removing a verifier as a random user: , 0x0000000000000000000000000000000000000539) [staticcall]
    │   └─ ← ()
    ├─ [0] VM::prank(0x0000000000000000000000000000000000000539)
    │   └─ ← ()
    ├─ [2809] VerifierList::deleteVerifier(test)
    │   └─ ← ()
    ├─ [0] console::log(getting the verifier with id: , test) [staticcall]
    │   └─ ← ()
    ├─ [4722] VerifierList::getVerifier(test) [staticcall]
    │   └─ ← , , , , []
    ├─ [0] console::log(fetched verifier id: , ) [staticcall]
    │   └─ ← ()
    ├─ [0] console::log(getting key and value for Verifier 'test' and check at index 'test') [staticcall]
    │   └─ ← ()
    ├─ [3549] VerifierList::getVerifierCheck(test, 0) [staticcall]
    │   └─ ← test, test
    ├─ [0] console::log(key: , test) [staticcall]
    │   └─ ← ()
    ├─ [0] console::log(value: , test) [staticcall]
    │   └─ ← ()
    └─ ← ()

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.76ms
Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:U (5.0)

Recommendation:

To fully purge the `checks` mapping, consider implementing a `deleteCheck` function, which would delete a particular value for the provided key.

Remediation Plan:

SOLVED: The Haqq team solved this issue in commit `9fcd3e1`. The `Verifier` struct now includes the `checksSize` field which stores the number of `Checks` for a given `Verifier` and the `deleteVerifier` function iterates over all `Checks` and deletes them before deleting the `Verifier`.

4.3 (HAL-03) ZERO ADDRESS AND ZERO LENGTH CHECKS MISSING - LOW (2.5)

Description:

Multiple setter functions in the VerifierList and RelayerList contracts expect the caller to provide data of the address and string types. Those functions fail to verify if the user-supplied addresses are not equal to zero address, and if the provided strings are not empty. This may lead to the contract storing corrupted or unusable data, which may affect the normal operations of the Haqq Social network.

Code Location:

Listing 4: contracts/contracts/VerifierList.sol (Lines 58-61)

```
47 function addVerifier(
48     string memory _id,
49     string memory _name,
50     string memory _jwk_url,
51     string memory _verifier,
52     Check[] memory _checks
53 ) external {
54     require(msg.sender == owner(), "Only admin node can set data")
↳ ;
55     require(keccak256(abi.encodePacked(verifiers[_id].id)) ==
↳ keccak256(abi.encodePacked("")), "Verifier already exists");
56
57     Verifier storage v = verifiers[_id];
58     v.id = _id;
59     v.name = _name;
60     v.jwk_url = _jwk_url;
61     v.verifier = _verifier;
```

Listing 5: contracts/contracts/RelayerList.sol (Lines 32,34)

```
29 constructor (address[] memory _relayerWallets, string[] memory
↳ _relayerAddress) {
```

```
30      require(_relayerWallets.length >= 5, "At least 5 relayer
↳ addresses are required.");
31      require(_relayerWallets.length == _relayerAddress.length, "
↳ Relayer wallet and address length mismatch.");
32      relayerAddresses = _relayerWallets;
33      for (uint256 i = 0; i < _relayerWallets.length; i++) {
34          _relayers[_relayerWallets[i]] = Relayer(_relayerWallets[i
↳ ], _relayerAddress[i]);
35      }
36  }
```

Listing 6: contracts/contracts/RelayerList.sol (Line 40)

```
38  function addRelayer(string memory _relayerAddress, address
↳ _walletAddress) external onlyOwner {
39      require(_relayers[_walletAddress].walletAddress == address(0),
↳  "Relayer already exists.");
40      _relayers[_walletAddress] = Relayer(_walletAddress,
↳ _relayerAddress);
41  }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

Consider validating the user supplied addresses are not equal to the zero address and that strings are not empty.

Remediation Plan:

SOLVED: The Haqq team solved this issue in commit 4ef69aa. Addresses, strings and byte arrays are checked not to be zero or of zero length.

4.4 (HAL-04) REDUNDANT STORAGE DATA - LOW (2.5)

Description:

In the VerifierList, the verifiers mapping stores Verifer data and is keyed by Verifier ID. One of the fields in the Verifier struct in the VerifierList contract is string id. Similarly, in the RelayerList contract, the _socialIDEntries mapping stores SocialID data. The SocialIDEntry struct stored in this mapping has a field called internalID of type string.

Verifiers in the verifier mapping are keyed by their IDs and SocialIDEntry instances in the _socialIDEntries mapping are keyed by their IDs. In both cases, mapping keys are repeated in mapping values, increasing gas consumption unnecessarily.

Code Location:

Listing 7: contracts/contracts/VerifierList.sol (Line 9)

```
 8 struct Verifier {
 9     string id;
10     string name;
11     string jwk_url;
12     string verifier;
13     mapping(uint => Check) checks;
14     uint checksSize;
15 }
```

Listing 8: contracts/contracts/VerifierList.sol (Lines 48,58)

```
47 function addVerifier(
48     string memory _id,
49     string memory _name,
50     string memory _jwk_url,
51     string memory _verifier,
52     Check[] memory _checks
```

```
53 ) external {
54     require(msg.sender == owner(), "Only admin node can set data")
↳ ;
55     require(keccak256(abi.encodePacked(verifiers[_id].id)) ==
↳ keccak256(abi.encodePacked("")), "Verifier already exists");
56
57     Verifier storage v = verifiers[_id];
58     v.id = _id;
```

Listing 9: contracts/contracts/RelayerList.sol (Line 15)

```
14 struct SocialIDEntry {
15     bytes32 internalID;
16     bytes32[] shareIndices;
17     address[] relayers;
18 }
```

Listing 10: contracts/contracts/RelayerList.sol (Lines 55,70)

```
55 function createRelayerList(bytes32 socialIDHash) external {
56     address[] memory copiedRelayerAddresses = new address[](
↳ relayerAddresses.length);
57     for (uint256 i = 0; i < relayerAddresses.length; i++) {
58         copiedRelayerAddresses[i] = relayerAddresses[i];
59     }
60
61     uint256 copiedLength = relayerAddresses.length;
62     address[] memory randomRelayers = new address[](5);
63     bytes32[] memory randomShares = new bytes32[](5);
64
65     for (uint256 i = 0; i < 5; i++) {
66         (randomRelayers[i], copiedLength) = _pickAndRemoveRandom(
↳ copiedRelayerAddresses, copiedLength);
67         randomShares[i] = _generateRandomShare(randomRelayers[i]);
68     }
69
70     _socialIDEntries[socialIDHash] = SocialIDEntry(socialIDHash,
↳ randomShares, randomRelayers);
71 }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

Consider removing duplicate parameters from both structs. Please note, implementing this change may require modifying the assertions in the `deleteVerifier` and `getRelayerList` functions, and the boolean expression returned by the `hasRelayerList` function.

Remediation Plan:

SOLVED: The Haqq team solved this issue in commit `9fcd3e1`. The duplicated parameters were removed from the `Verifier` and `SocialIDEntry` structs.

4.5 (HAL-05) SINGLE STEP OWNERSHIP TRANSFER PROCESS - INFORMATIONAL (0.7)

Description:

The RelayerList, RelayerStorage and VerifierList contracts inherit from the Ownable contract by OpenZeppelin. One of the features offered by this contract is a single-step ownership transfer process. What this means is the current contract owner can transfer contract ownership to another address directly. If the new owner is not willing to assume the role or the address does not yet exist, access to privileged functions in the contract may be lost.

Code Location:

Listing 11: contracts/contracts/RelayerList.sol (Line 7)

```
5 import "@openzeppelin/contracts/access/Ownable.sol";
6
7 contract RelayerList is Ownable {
```

Listing 12: contracts/contracts/RelayerStorage.sol (Line 7)

```
4 import "@openzeppelin/contracts/access/Ownable.sol";
5 import "@openzeppelin/contracts/security/Pausable.sol";
6
7 contract RelayStorage is Ownable, Pausable {
```

Listing 13: contracts/contracts/RelayerStorage.sol (Line 6)

```
4 import "@openzeppelin/contracts/access/Ownable.sol";
5
6 contract VerifierList is Ownable {
```

BVSS:

AO:S/AC:L/AX:H/C:N/I:C/A:N/D:N/Y:N/R:N/S:U (0.7)

Recommendation:

Consider inheriting from the Ownable2Step to require the new owner confirm the nomination before it is actually transferred.

Remediation Plan:

SOLVED: The Haqq team solved this issue in commit 9fcd3e1.

4.6 (HAL-06) IMMUTABLE RELAYER ADDRESSES ARRAY - INFORMATIONAL (0.7)

Description:

The relayerAddresses array in the RelayerList contract stores the wallets of the Relayers registered on contract creation. This array is immutable, and if any of the wallets is compromised it cannot be removed from the array. This may pose a threat to normal contract operations and the broader network.

Code Location:

Listing 14: contracts/contracts/RelayerList.sol (Lines 32,34)

```
29 constructor (address[] memory _relayerWallets, string[] memory
↳ _relayerAddress) {
30     require(_relayerWallets.length >= 5, "At least 5 relayer
↳ addresses are required.");
31     require(_relayerWallets.length == _relayerAddress.length, "
↳ Relayer wallet and address length mismatch.");
32     relayerAddresses = _relayerWallets;
33     for (uint256 i = 0; i < _relayerWallets.length; i++) {
34         _relayers[_relayerWallets[i]] = Relayer(_relayerWallets[i
↳ ], _relayerAddress[i]);
35     }
36 }
```

BVSS:

AO:S/AC:L/AX:H/C:N/I:C/A:N/D:N/Y:N/R:N/S:U (0.7)

Recommendation:

Consider implementing a `removeRelayer` which can be used to remove a `Relayer` address from the `relayerAddresses` array.

Remediation Plan:

SOLVED: The Haqq team solved this issue in commit `e6c9db4`. `Relayers` can be removed from the `relayerAddresses` array with the `removeRelayer` function.

4.7 (HAL-07) REDUNDANT COMPUTATION INSIDE LOOP - INFORMATIONAL (0.0)

Description:

In the RelayerList contract, the createRelayerList function selects Relayers for a Social ID by generating a pseudo random value and using it as the index of a Relayer to select from the relayerAddresses array. This pseudo random value is an uint256 parsed from a keccak256 hash of the following environment variables:

- block.timestamp
- block.prevrandao
- msg.sender

Those variables are constant in a function call and so the pseudo random value is constant in every iteration of the loop, therefore calculating it in the loop unnecessarily increases gas consumption by over 200 per iteration.

Code Location:

Listing 15: contracts/contracts/RelayerList.sol (Line 9)

```
8 for (uint256 i = 0; i < 5; i++) {
9     (randomRelayers[i], copiedLength) = _pickAndRemoveRandom(
↳ copiedRelayerAddresses, copiedLength);
10     randomShares[i] = _generateRandomShare(randomRelayers[i]);
11 }
```

Listing 16: contracts/contracts/VerifierList.sol (Line 49)

```
48 function _pickAndRemoveRandom(address[] memory array, uint256
↳ length) private view returns (address, uint256) {
49     uint256 randomIndex = uint256(keccak256(abi.encodePacked(block
↳ .timestamp, block.prevrandao, msg.sender))) % length;
50     address picked = array[randomIndex];
```

```
51      array[randomIndex] = array[length - 1];
52      return (picked, length - 1);
53  }
```

Proof of Concept:

Listing 17

```
1  function testGasConsumptionKeccak() public view {
2      uint256(keccak256(abi.encodePacked(block.timestamp, block.
↳  prevrandao, msg.sender)));
3  }
4
5  function testGasConsumptionKeccakEmpty() public view {
6
7  }
```

```
Running 2 tests for test/Security.t.sol:VerifierListTest
[PASS] testGasConsumptionKeccak() (gas: 414)
[PASS] testGasConsumptionKeccakEmpty() (gas: 209)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 3.20ms
Ran 1 test suites: 2 tests passed, 0 failed, 0 skipped (2 total tests)
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

Consider moving the calculating of the intermediary pseudo random value outside the `for` loop. Please note, implementing this change requires modifying the `_pickAndRemoveRandom` function arguments list to include the generated value.

Remediation Plan:

SOLVED: The Haqq team solved this issue in commit 9fcd3e1. The calculation of the pseudo random value was moved outside the for loop.

4.8 (HAL-08) INEFFICTIENT FOR LOOPS - INFORMATIONAL (0.0)

Description:

For loops are used in several functions in the contracts in scope. It was identified that all loops in the contract can be optimized to make the contracts more gas efficient:

- when iterating over an array, cache the array length outside of loops to avoid reading from memory/storage and pushing the length to stack in every iteration,
- when iterating from the 0 index, do not initialize the index to 0 because all numerical values in Solidity are initialized to zero by default,
- when incrementing the index, use preincrementation instead of postincrementation, and do it in an unchecked block.

Code Location:

Listing 18: contracts/contracts/RelayerList.sol (Line 33)

```
33 for (uint256 i = 0; i < _relayerWallets.length; i++) {
34     _relayers[_relayerWallets[i]] = Relayer(_relayerWallets[i],
↳ _relayerAddress[i]);
35 }
```

Listing 19: contracts/contracts/RelayerList.sol (Line 57)

```
57 for (uint256 i = 0; i < relayerAddresses.length; i++) {
58     copiedRelayerAddresses[i] = relayerAddresses[i];
59 }
```

Listing 20: contracts/contracts/RelayerList.sol (Line 65)

```
65 for (uint256 i = 0; i < 5; i++) {
66     (randomRelayers[i], copiedLength) = _pickAndRemoveRandom(
```

```
↳ copiedRelayerAddresses, copiedLength);
67      randomShares[i] = _generateRandomShare(randomRelayers[i]);
68  }
```

Listing 21: contracts/contracts/RelayerList.sol (Line 81)

```
81  for (uint256 i = 0; i < entry.relayers.length; i++) {
82      relayerData[i] = RelayerData(_relayers[entry.relayers[i]].
↳ relayerAddress, entry.shareIndices[i]);
83  }
```

Listing 22: contracts/contracts/VerifierList.sol (Line 38)

```
37  for (uint i = 0; i < v.checksSize; i++) {
38      checksArray[i] = Check(v.checks[i].key, v.checks[i].value);
39  }
```

Listing 23: contracts/contracts/VerifierList.sol (Line 64)

```
64  for (uint i = 0; i < _checks.length; i++) {
65      v.checks[v.checksSize] = Check(_checks[i].key, _checks[i].
↳ value);
66      v.checksSize++;
67  }
```

Proof of Concept:

Listing 24

```
1  function testGasConsumptionForUnoptimized() public pure {
2      uint256[5] memory array;
3
4      for (uint256 i = 0; i < array.length; i++) { }
5  }
6
7  function testGasConsumptionForOptimized() public pure {
8      uint256[5] memory array;
9      uint256 arrayLength = array.length;
10
11     for (uint256 i; i < arrayLength;) {
```

```
12          unchecked {
13          ++i;
14          }
15      }
16  }
```

```
Running 2 tests for test/Security.t.sol:VerifierListTest
[PASS] testGasConsumptionForOptimized() (gas: 590)
[PASS] testGasConsumptionForUnoptimized() (gas: 903)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 3.32ms
Ran 1 test suites: 2 tests passed, 0 failed, 0 skipped (2 total tests)
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

Consider using the following pattern for for loops:

Listing 25

```
1 uint256 arrayLength = array.length;
2
3 for (uint256 i; i < arrayLength;) {
4     . . .
5     unchecked {
6      ++i
7     }
8 }
```

Remediation Plan:

ACKNOWLEDGED: The Haqq team acknowledged this finding and applied some gas optimization techniques up to commit 55e33fd.

AUTOMATED TESTING

5.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIs and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.
The security team assessed all findings identified by the Slither software, however, findings with severity Information and Optimization are not included in the below results for the sake of report readability.

Results:

Slither results for RelayerList.sol	
Finding	**Impact**
RelayerList._pickAndRemoveRandom(address[],uint256) (src/RelayerList.sol#48-53) uses a weak PRNG: "randomIndex = uint256(keccak256(bytes)(abi.encodePacked(block.timestamp,block.prevrandao,msg.sender))) % length (src/RelayerList.sol#49)"	High
End of table for RelayerList.sol	

Because of the purpose of this function (selecting a Relayer) this finding can be considered a false positive.

Slither results for VerifierList.sol	
Finding	**Impact**
VerifierList.deleteVerifier(string) (src/VerifierList.sol#70-73) deletes VerifierList.Verifier (src/VerifierList.sol#7-14) which contains a mapping: -delete verifiers[_id] (src/VerifierList.sol#72)	Medium
End of table for VerifierList.sol	

Please see HAL-02.

No issues were identified in the RelayerStorage.sol file.

Results summary:

In the RelayerList contract, because of the purpose of this function (selecting a Relayer) the weak PNG generator finding can be considered a false positive.

For more details on the VerifierList contract delete finding, please refer to HAL-02.

5.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers in order to locate any vulnerabilities.

Results:

No issues were detected by MythX.

THANK YOU FOR CHOOSING

// HALBORN