Aramid Bridge Code Review

Draft Findings and Recommendations Report Presented to:

Aramid Finance

January 18, 2023
Version: 1.2

Presented by:

NAGRAVISION SÀRL
Route de Genève 22-24
1033 Cheseaux-sur-Lausanne
Switzerland

FOR PUBLIC RELEASE

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

Overview

Aramid Finance engaged Kudelski Security to perform a secure code assessment of the soldier app, smart contracts and web app for the Aramid token bridge.

The assessment was conducted remotely by the Kudelski Security Team.
Testing took place on 22nd September - 28th October, 2022, and focused on the following objectives:

- Provide the customer with an assessment of their overall security posture and any risks that were discovered with the smart contracts.
- To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures that are in place.
- To identify potential issues and include improvement recommendations based on the result of our tests.

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the Kudelski Security Teams took to identify and validate each issue, as well as any applicable recommendations for remediation.

Key findings

The following are the major themes and issues identified during the testing period. These, along with other items, within the findings section, should be prioritized for remediation to reduce to the risk they pose.

- Missed validation checks – the Ethereum smart contracts relied on the signature of soldiers for various operations. These include releasing funds, adding and removing soldiers and tokens, changing signature threshold and updating contracts. It was observed that smart contracts were not verifying all the signatures passed to signature validation function and could result in an incorrectly failing signature.

During the code review, the following positive observations were noted regarding the scope of the engagement:

- The code was clean in general
- Extensive input validation was done in the code to make sure transaction is only processed if all arguments are verified by soldiers.
- Tests were adequate.
- Engagement with the technical teams was strong, enriching, and responsive, which is significant for performing a security review.

Scope and Rules of Engagement

Kudelski Security performed a code review of the soldier application, smart contracts and web app for Aramid Finance. The following table documents the targets in scope for the engagement. No additional systems or resources were in scope for this assessment.

The source code was supplied with the commit hashes in private repositories at:

- https://github.com/AramidFinance/bridge-soldier-nodejs-app/commit/ad26b5d538fcea26da256896de6454c2970e26ef
- https://github.com/AramidFinance/bridge-ethereum-assets/commit/1459d784be2455a0191b0c388807920e4b9a43c5
- https://github.com/AramidFinance/bridge-web-app/commit/272385e9403844416300b911a98d71df84cdb2db

A further round of review was conducted on 4th January, 2023 following remediations on the following commit hashes.

- https://github.com/AramidFinance/bridge-ethereum-assets/tree/b1eca7157dd287bc07c84ba4405be0f4b2be2698

In-Scope Code	
bridge-soldier-nodejs-app	bridge-ethereum-assets
soldier/ ├── algo ├── algo2algo ├── algo2eth ├── common ├── eth ├── eth2algo ├── eth2eth ├── interface ├── ipfs ├── p2p ├── store └── main.ts	├── contracts │ ├── AramidAlgoToken.sol │ ├── AramidAlgoTokenMainnet.sol │ ├── AramidAuroraToken.sol │ ├── AramidAuroraTokenMainnet.sol │ ├── AramidBitcoinToken.sol │ ├── AramidBitcoinTokenMainnet.sol │ ├── AramidDaoToken.sol │ ├── AramidEthTokenMainnet.sol │ ├── AramidMumbaiToken.sol │ ├── AramidPolygonTokenMainnet.sol │ ├── AramidRinkebyToken.sol │ ├── AramidUSDToken.sol │ ├── AramidUSDTokenMainnet.sol │ ├── Bridge.sol │ ├── BridgeGovernance.sol │ ├── BridgeProxy.sol │ ├── BridgeSignatureValidator.sol │ ├── BridgeState.sol │ ├── Migrations.sol │ ├── WrappedAssetToken.sol │ ├── WrappedAssetTokenMintable.sol
bridge-web-app	

Table 1: Scope

TECHNICAL ANALYSIS & FINDINGS

During the Aramid Bridge Code Review, we discovered 2 findings that had a medium severity rating, as well as 3 of low severity.

The following chart displays the findings by severity.

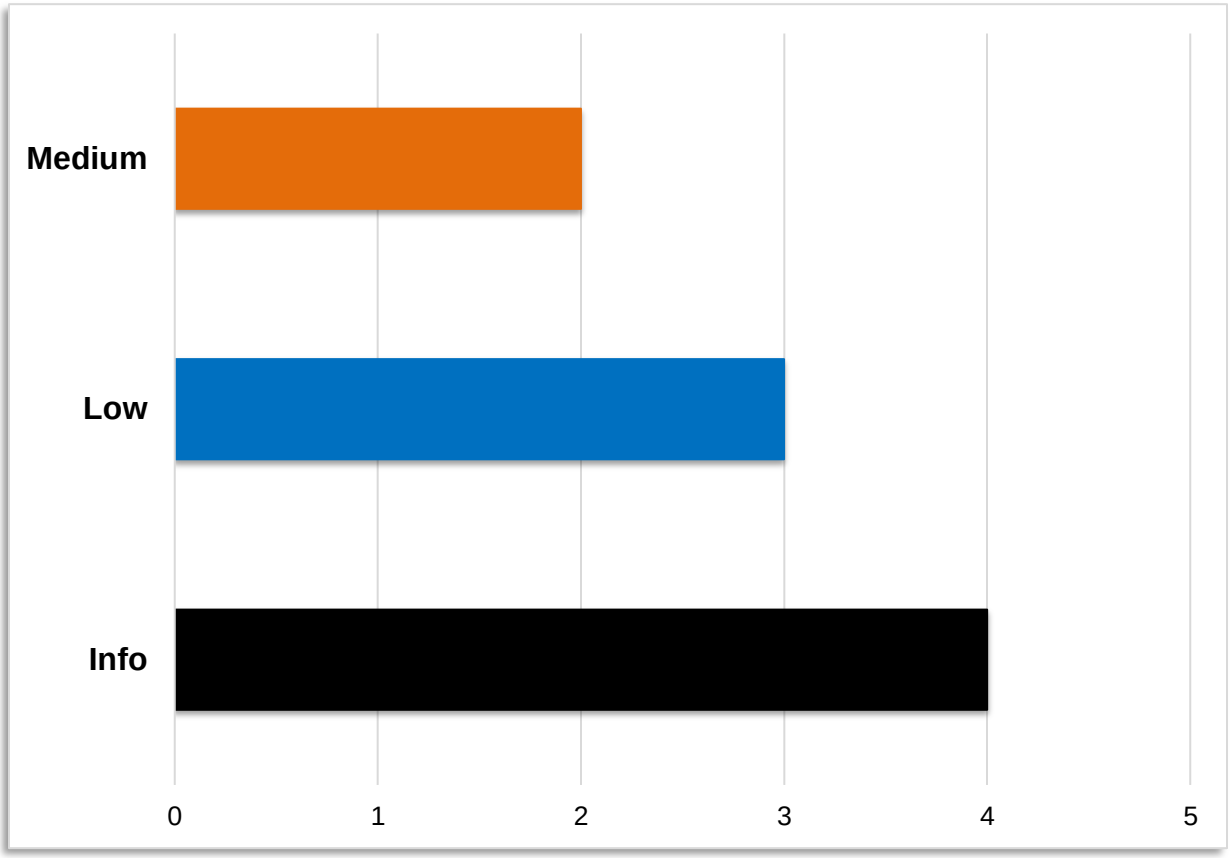

Figure 1: Findings by Severity

Findings

The *Findings* section provides detailed information on each of the findings, including methods of discovery, explanation of severity determination, recommendations, and applicable references.

The following table provides an overview of the findings.

#	Severity	Description	Status
Ethereum Contracts			
KS-ARDC-01	Medium	One incorrect signature could result in rejected transaction	Remediated
KS-ARDC-02	Medium	Minimum signature threshold not set in Ethereum contracts	Remediated
KS-ARDC-03	Low	Max Bridge fee not implemented	Acknowledged
KS-ARDC-04	Informational	Use of old Solidity version	Remediated
Soldier App			
KS-ARDS-01	Low	Hardcoded values in code	Acknowledged
KS-ARDS-02	Low	Transaction cannot be reprocessed If release token fails	Partially Remediated
KS-ARDS-03	Informational	Potential duplication of executioner processing	Remediated
KS-ARDS-04	Informational	Potential functionality description in TODO	Remediated
KS-ARDS-05	Informational	Outdated/unused/dead code to cleanup/remove	Acknowledged

Table 2: Findings Overview

KS-ARDC-01 – One incorrect signature could result in rejected transaction

Severity	MEDIUM
Status	REMEDIATED

Impact	Likelihood	Difficulty
Low	Medium	Moderate

Description

A set of off-chain nodes, called soldiers, watch token lock transactions, validate them, and issue corresponding token release multisig transactions in the destination network and to the destination address signalled in the token lock transaction. Soldiers exchange the transactions among them over a p2p protocol, validate the transactions, and add their signatures. Release token transactions are multisig, requiring N out of M signatures in order to be accepted by the destination chain.

It was observed that contrary to documentation the code was verifying only T out of T signatures.

Impact

`Signature.length` refers to the number of signatories of a particular transaction, while threshold variable contains the minimum number of valid signatures required to pass the transaction. In the `validateSignatures()` function only signatures until the threshold are being validated, while the rest of the signatures are ignored. This means, even one invalid signature would invalidate the transaction, which is against the logic of threshold signature.

```
contract BridgeSignatureValidator is BridgeState {
    function validateSignatures(
        bytes32 message,
        bytes[] memory signatures,
        mapping(address => bool) storage signedBy
    ) internal {
        require(
            signatures.length >= signaturesThreshold,
            "Not enough signatures"
        );
        for (uint256 i = 0; i < signaturesThreshold; i++) {
            address recoveredAddress = recoverSigner(message, signatures[i]);

            require(soldiers[recoveredAddress], "Invalid signature");
            require(!signedBy[recoveredAddress], "Duplicated signature");

            signedBy[recoveredAddress] = true;
        }
    }
```

Affected Resources

- `BridgeSignatureValidator.sol`
- `Bridge.sol`

- `BridgeGovernance.sol`

Recommendation

As per the code it is possible that signatures passed to `validateSignatures()` function are more than the set threshold. However, only signatures until the set threshold are being validated and rest are not being validated.

It is recommended that loop should run until the `signatures.length` and each successful verification should be counted until threshold is reached. This means that `require` condition (line 19) need to be removed as it is possible for a signature verification to fail. `Require` condition should only be implemented to make sure that signature is not reused as well as that number of verified signatures are greater than or equal to threshold.

This is a very important function and is being used to add/remove tokens, add/remove soliders and validate transactions. We recommend the following code for the `validateSignatures()` function.

```
function validateSignatures(
    bytes32 message,
    bytes[] memory signatures,
    mapping(address => bool) storage signedBy
) internal {
    require(
        signatures.length >= signaturesThreshold,
        "Not enough signatures"
    );
    uint8 thresholdcount = 0;
    for (uint256 i = 0; i < signatures.length; i++) {
        address recoveredAddress = recoverSigner(message, signatures[i]);
        if (soldiers[recoveredAddress] == true) {
            thresholdcount++;
        }
        //require(soldiers[recoveredAddress], "Invalid signature");
        require(!signedBy[recoveredAddress], "Duplicated signature");

        signedBy[recoveredAddress] = true;
    }
    require(thresholdcount >= signaturesThreshold);
}
```

This function ensures that we check all the signatures passed into the function as well as that function executes successfully if and only if number of verified signatures are equal to or more than the threshold.

KS-ARDC-02 – Minimum signature threshold not set in Ethereum contracts

Severity	MEDIUM
Status	REMEDIATED

Impact	Likelihood	Difficulty
High	Low	Hard

Description

Aramid bridge relies extensively on soliders who verify the transactions before a user can execute the transaction to release the tokens. Soldiers are also used extensively for governance purpose. Ethereum does not have default support of threshold signature. As a result, Aramid implemented its own version of threshold signatures where smart contract verifies if a certain transaction is signed by at least certain number of soldiers.

It was observed that there is no minimum threshold set for number of signatures required to perform a transaction.

Impact

It is possible to set threshold to as low as 1 either intentionally or accidentally, therefore making the threshold signature practically invalid.

Affected Resources

- `BridgeGovernance.sol`

Evidence

```
function changeSignatureThreshold(
      uint8 threshold,
      uint256 requestNonce,
      bytes[] memory signatures
   ) public onlySoldier validNonce(requestNonce) {
      validateSignatures(
         reconstructMessageUint8(threshold, requestNonce),
         signatures,
         signers[requestNonce]
      );
      signaturesThreshold = threshold;
      emit LogThresholdChanged(threshold);
   }
```

Recommendation

It is recommended to set the minimum threshold to at least 50% or more of all soldiers.

KS-ARDC-03 – Max Bridge fee not implemented

Severity	LOW
Status	ACKNOWLEDGED

Impact	Likelihood	Difficulty
Low	Low	Hard

Description

Aramid charges bridge fee to the users for the transaction. This amount is calculated by the soldier app and is locked on the Ethereum blockchain using `Locktokens`() function in `bridge.sol` contract. It was observed that although the smart contract verifies if the bridge fee is greater than 0, however there is no check on max bridge fee.

Impact

It is possible that, whether maliciously or due to market conditions, the bridge fee could become unfeasibly high for users to transact.

Affected Resources

- `Bridge.sol`

Evidence

```
if (
        rootTokenAddr != feeTokenAddr &&
        feeAmount > 0 &&
        feeTokenAddr != address(0)
    ) {
```

```
else if (rootTokenAddr == feeTokenAddr && feeAmount > 0) {
        // 2 - Token used to pay fee is the same as token to bridge && fee > 0
        IERC20(rootTokenAddr).transferFrom(
            msg.sender,
            address(this),
            rootAmount + feeAmount
        );
```

Recommendation

It is recommended to set the maximum fee amount and check the fee amount that is being locked in the contract against the maximum fee.

KS-ARDC-04 – Use of old Solidity version

Severity	INFORMATIONAL
Status	REMEDIATED

Impact	Likelihood	Difficulty
-	-	-

Description

Outdated or weak components are in use by the application. These components may be part of a programming library or underlying platform. These weaknesses are commonly targeted by attackers because of the publicly available information on these vulnerabilities. It was observed that Migration.sol allows versions with known vulnerabilities. Similarly, all other contracts use pragma versions ^0.8.0, which should also be updated to latest known good version.

Affected Resource

- Migration.sol

Evidence

Known vulnerabilities for Solidity version 0.8 prior to 0.8.16.
https://docs.soliditylang.org/en/v0.8.16/bugs.html

```
pragma solidity >=0.4.22 <0.9.0;
```

Recommendation

File should be upgraded to use latest known good version.

KS-ARDS-01 – Hardcoded values in code

Severity	LOW
Status	ACKNOWLEDGED

Impact	Likelihood	Difficulty
Low	Low	Easy

Description

Specific system address was discovered in the code.
If such a code is public, this value is publicly readable by anyone.

Impact

Attackers may leverage this address to attempt to disrupt the normal traffic of the targeted service by overwhelming the target or its surrounding infrastructure with a flood of Internet traffic (DDoS).

Evidence

```
22  const createNode = async () => {
23    let bootstrapers = ['/ip4/[redacted]/tcp/[redacted]/p2p/[redacted]'];
24
```

Hardcoded address

Affected Resource

- p2p/createNode.js (Line 23)

Recommendation

By default, never store sensitive information in code.
All data relative to deployment/management activities should be stored in private/local configuration files.

Reference

N/A

KS-ARDS-02 – Transaction cannot be reprocessed If release token fails

Severity	LOW
Status	PARTIALLY REMEDIATED

Impact	Likelihood	Difficulty
Low	Medium	Hard

Description

Aramid bridge uses the lock and release mechanism to lock the funds on one blockchain and release equivalent tokens (1:1 Mapping) on the other blockchain. The soldier apps continuously scan these blockchains for new transactions and when a new transaction occurs on one blockchain, the soldiers validate and process the transaction and the executioner (randomly selected among soldier nodes) submits the transaction on the other blockchain. This transaction contains the signatures of the soldiers and metadata of the transaction. User then uses this transaction to get the funds released on the blockchain.

It was observed that soldiers set the transaction status as “Processed” upon submission to the blockchain. Therefore, if the transaction fails due to invalid signatures then it cannot be reprocessed by the soldiers as for the soldiers transaction status will be shown as “Processed”. This means the user will not be able to get the tokens released. This can also happen if user is unable to submit the transaction due to failure of internet connection or any other reasons and the max round until which the user should have submitted the transaction expires.

Impact

User will not be able to get tokens released and it would need to be processed manually.

Evidence

```
248         };
249         state = newState;
250         await setState(state);
251       } else {
252         if (state.state === StateTypeEnum.Submitting) {
253           if (moment(state.time) < moment().subtract(2, 'minutes')) {
254             // if we submitted more than 2 minutes ago, and we are still in submission process now,
255             state.time = new Date();
256             await setState(state);
257           } else {
258             return false;
259           }
260         } else if (state.state === StateTypeEnum.Processed) {
261           return false; //already submitted
262         }
263       }
```

Transaction cannot be reprocessed if submitted

Affected Resources

- `p2p/process/processSignedPayloadMessage.ts (Lines 249-262)`

Recommendation

Implement a function in soldier app to query the smart contract (on Ethereum blockchain) to check the status of processed transactions to make sure that all transaction IDs signed by soldiers are processed. Similar function can be implemented for Algorand blockchain to check the status of certain transaction.

KS-ARDS-03 – Potential duplication of executioner processing

Severity	INFORMATIONAL
Status	REMEDIATED

Impact	Likelihood	Difficulty
-	-	-

Description

An executioner is a soldier node that is randomly selected to submit a validated transaction to the blockchain. Aramid selects the executioner for a transaction each 60 seconds. This selection is based on the following code.

```
const executionerIndex = (parseInt(T) + parseInt(removeNonNumbers)) %
addrs.length;

Where T  = new BigNumber(time.unix()).dividedBy(60).toFixed(0, 1);
RemoveNonNumbers = sourceTransactionId.replace(/[^\d.-]/g, '');
And address.length = number of soldiers.
```

Based on the public configuration parameters, It is possible that a soldiers take more than a minute to validate, sign and submit the transaction to the blockchain. Therefore, two soldier nodes may get selected as executioner.

This can happen if the time required to wait for the number of confirmations (rounds passed between the time when transaction was inserted into blockchain and current round) is higher than the time when a new executioner is selected. The public configuration file in the review required 3 rounds to have passed between the round in which transaction was inserted and current round. Ethereum blockchain has inter block time of 12 seconds. This means it was possible that 60 seconds will pass during the transaction validation, signing and submission and as such, a new executioner could resubmit the transaction.

Impact:

Submission of duplicated transaction will not impact the user funds as validity checks for duplicate processing are implemented. However, this additional processing of transaction should be avoided.

Evidence

```
14      }
15      const T = new BigNumber(time.unix()).dividedBy(60).toFixed(0, 1); // every 60 seconds there
16      let addrs = await getSoldiersByRound(destinationChainId, destinationRound);
17      let removeNonNumbers = sourceTransactionId.replace(/[^\d.-]/g, '');
18      if (removeNonNumbers.length > 5) removeNonNumbers = removeNonNumbers.substring(0, 5);
19      const executionerIndex = (parseInt(T) + parseInt(removeNonNumbers)) % addrs.length;
20      return addrs[executionerIndex];
21    } catch (e) {
```

Executioner Selection

Affected Resources

- `common/getExecutioner.ts (Line 19)`

Recommendation

`getExecutioner` function should return same executioner for a transaction based on minimum time it would take to execute a transaction. Based on existing configuration, we recommend it to be based on the following equation:

```
(RequiredConfirmationsRounds  * interblocktime)
```

e.g. if required confirmations are 10 and interblock time is 12 seconds then there should be same executioner for at least 2 minutes.

KS-ARDS-04 – Potential functionality description in TODO

Severity	INFORMATIONAL
Status	REMEDIATED

Impact	Likelihood	Difficulty
-	-	-

Description

Kudelski Security observed that the code has TODO pointing out that the implementation of checks isn't complete yet and needs to be implemented.

Impact

Describing missing logic could be used as a part of a sophisticated attack where missing functionality could be used to crash or extract information from the application.

Evidence

```
61          case ChainTypeEnum.algo:
62            // Load algorand tx and verify that the tx original txid is equal to this proof
63            // TODO .. more checks
64            const tx = await getAlgorandTransaction(proofPayload.proof.destinationTransaction,
```

Checks not yet implemented

Affected Resource

- p2p/process/processProofMessage.ts (Line 63)

Recommendation

This should be implemented or removed.

Reference

N/A

KS-ARDS-05 – Outdated/unused/dead code to cleanup/remove

Severity	INFORMATIONAL
Status	ACKNOWLEDGED

Impact	Likelihood	Difficulty
-	-	-

Description

Kudelski Security observed that a parameter is not used in a function. Either this parameter is necessary - and should be used according to its purpose or it should be removed from the function signature (and from all calling code)

Impact

Such code not being aligned with its documentation confuses both developers and reviewers. Dead code that results from code that can never be executed is an indication of problems with the source code that needs to be fixed and is an indication of poor quality.

Evidence

```
 9  /**
10   * Sends signed payload to other soldiers
11   *
12   * @param {*} node - The libp2p node instance connected to a AramidBridge gossip network.
13   * @param {*} channel - The topic the node is subscribed to.
14   * @param {*} payload - payload
15   * @param {*} doNotResubmit - Do not resubmit to the blockchain if i have already signed before this payload
16   */
17  const sendPayload = async (node: Libp2p, channel: string, payload: Payload, doNotResubmit: boolean) => {
18    const logger = getLogger();
...
30      }
31      if (appConfiguration.useBlockchainComm) {
32        //!doNotResubmit &&
33        var publicConfiguration = await getPublicConfiguration(false);
```

Unused parameter: doNotResubmit

Affected Resource

- Callee
  - p2p/commands/sendPayload.ts (Lines 14, 17, 32)
- Caller
  - algo/message/processITransfer.ts (Line 68)
  - eth/watchEthEvents.ts (Line 71)
  - p2p/commands/sendPayload.ts (Line 17)
  - p2p/process/processSignedPayloadMessage.ts (Line 177, 362)
  - timer/trackUnprocessedPayloads.ts (Line 49)

Recommendation

As this parameter no longer seemed necessary, the code should be cleaned up.

Reference

https://cwe.mitre.org/data/definitions/561.html

METHODOLOGY

During this source code review, the Kudelski Security Services team reviewed code within the project within an appropriate IDE. During every review, the team spends considerable time working with the client to determine correct and expected functionality, business logic, and content to ensure that findings incorporate this business logic into each description and impact. Following this discovery phase the team works through the following categories:

- Authentication
- Authorization and Access Control
- Auditing and Logging
- Injection and Tampering
- Configuration Issues
- Logic Flaws
- Cryptography

These categories incorporate common vulnerabilities such as the OWASP Top 10

Tools

The following tools were used during this portion of the test. A link for more information about the tool is provided as well.

- Visual Studio Code
- Node
- Slither
- Mythril
- Solgraph

Vulnerability Scoring Systems

Kudelski Security utilizes a vulnerability scoring system based on impact of the vulnerability, likelihood of an attack against the vulnerability, and the difficulty of executing an attack against the vulnerability based on a high, medium, and low rating system

Impact
The overall effect of the vulnerability against the system or organization based on the areas of concern or affected components discussed with the client during the scoping of the engagement.

> **High:**
> The vulnerability has a severe effect on the company and systems or has an affect within one of the primary areas of concern noted by the client
>
> **Medium:**
> It is reasonable to assume that the vulnerability would have a measurable effect on the company and systems that may cause minor financial or reputational damage.
>
> **Low:**
> There is little to no affect from the vulnerability being compromised. These vulnerabilities could lead to complex attacks or create footholds used in more severe attacks.

Likelihood
The likelihood of an attacker discovering a vulnerability, exploiting it, and obtaining a foothold varies based on a variety of factors including compensating controls, location of the application, availability of commonly used exploits, and institutional knowledge

> **High:**
> It is extremely likely that this vulnerability will be discovered and abused
>
> **Medium:**
> It is likely that this vulnerability will be discovered and abused by a skilled attacker
>
> **Low:**
> It is unlikely that this vulnerability will be discovered or abused when discovered.

Difficulty
Difficulty is measured according to the ease of exploit by an attacker based on availability of readily available exploits, knowledge of the system, and complexity of attack. It should be noted that a LOW difficulty results in a HIGHER severity.

> **Easy:**
> The vulnerability is easy to exploit or has readily available techniques for exploit
>
> **Moderate:**
> The vulnerability is partially defended against, difficult to exploit, or requires a skilled attacker to exploit.
>
> **Hard:**
> The vulnerability is difficult to exploit and requires advanced knowledge from a skilled attacker to write an exploit

Severity
Severity is the overall score of the weakness or vulnerability as it is measured from Impact, Likelihood, and Difficulty

KUDELSKI SECURITY CONTACTS

NAME	POSITION	CONTACT INFORMATION
Jamshed Memon	**Blockchain Expert**	**jamshed.memon@kudelskisecurity.com**
Ronan Le Gallic	**Lead Engineer**	**ronan.legallic@nagra.com**