decurity

Smart Contract Security Audit Report

1inch Calldata Compressor

1. Contents

2. General Information

This report contains information about the results of the security audit of the 1inch (hereafter referred to as "Customer") Calldata Compressor smart contracts, conducted by Decurity in the period from 15/05/2023 to 19/05/2023.

2.1. Introduction

Tasks solved during the work are:

- Review the compression algorithm design and the usage of 3rd party dependencies,
- Audit the algorithm implementation,
- Develop the recommendations and suggestions to improve the security of the contracts.

2.2. Scope of Work

The audit scope included the smart contract DecompressorExtension.sol in 1inch/calldata-compressor repository. Initial review was done for the commit 508e4da767fb3ee95bcca843ecea0aff7d5fc38c.

2.3. Threat Model

The assessment presumes actions of an intruder who might have capabilities of any role (an external user, a contract owner, an external contract). The centralization risks have not been considered upon the request of the Customer.

The main possible threat actors are:

- User,
- Contract owner

The table below contains sample attacks that malicious attackers might carry out.

Table. Theoretically possible attacks

Attack	Actor
Contract code or data hijacking *Deploying a malicious contract or submitting malicious data*	Contract owner
Attacks on implementation *Exploiting the weaknesses in the compiler or the runtime of the smart contracts*	Anyone

2.4. Weakness Scoring

An expert evaluation scores the findings in this report, an impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

2.5. Disclaimer

Due to the intrinsic nature of the software and vulnerabilities and the changing threat landscape, it cannot be generally guaranteed that a certain security property of a program holds.

Therefore, this report is provided "as is" and is not a guarantee that the analyzed system does not contain any other security weaknesses or vulnerabilities. Furthermore, this report is not an endorsement of the Customer's project, nor is it an investment advice.

That being said, Decurity exercises best effort to perform their contractual obligations and follow the industry methodologies to discover as many weaknesses as possible and maximize the audit coverage using the limited resources.

3. Summary

As a result of this work, the compression/decompression algorithm and smart contract were proved to be solid.

3.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of May 19, 2023.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
Limited calldata decompression tests	test/Decompressor.js	Low	Not fixed

4. General Recommendations

This section contains general recommendations on how to improve overall security level. The Findings section contains technical recommendations for each discovered issue.

4.1. Security Process Improvement

The following is a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level:

- Keep the whitepaper and documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Perform regular audits for all the new contracts and updates,
- Ensure the secure off-chain storage and processing of the credentials (e.g. the privileged private keys),
- Launch a public bug bounty campaign for the contracts.

5. Findings

5.1. Limited calldata decompression tests

Risk Level: Low

Status: Not fixed

Files:

- test/Decompressor.js

Location: Function: "should decompress calldatas with all cases".

Description:

The hardhat test case `should decompress calldatas with all cases` checks the decompression algorithm using real world calldata samples loaded from `tx-calldata.json` available via IPFS (CID: `QmQnjk5MKMiC6jZNjWgXKkbCMFFznNYaHRsS281jV94wpe`). This file contains 1936 calldata samples, however the test case only uses the first five samples as indicated by `CALLDATAS_LIMIT` constant. This limitation makes this test just a sanity check. If the algorithm is modified and `CALLDATAS_LIMIT` is not increased, the new implementation will not get full coverage, which may lead to unforeseen vulnerabilities.

Remediation:

It is recommended to employ more comprehensive CI testing, which will cover all the calldata samples from `tx-calldata.json` on each smart contract update. This may not be feasible with Hardhat though as the testing of even 1936 samples takes too much time. Alternative more performant frameworks should be considered, such as Foundry. Besides, other testing approaches should be implemented, such as calldata fuzzing (see Appendix for a reference implementation).

6. Appendix

6.1. Fuzz testing example

This is an example of a calldata fuzzing implemented as a Hardhat test:

```
const hre = require('hardhat');
const { ethers } = hre;
const { loadFixture } =
require('@nomicfoundation/hardhat-network-helpers');
const { ether, expect, trim0x } = require('@1inch/solidity-utils');
const { compress } = require('../js/compressor.js');

let popularCalldatas = [];
try {
    popularCalldatas = require('./dict.json');
} catch (e) {}

describe('Decompressor', function () {
    async function initContracts () {
        const [addr1, addr2] = await ethers.getSigners();
        const chainId = (await ethers.provider.getNetwork()).chainId;

        const DecompressorExtMock = await
ethers.getContractFactory('DecompressorExtensionMock');
        const decompressorExt = await
DecompressorExtMock.deploy('TokenWithDecompressor', 'TWD');
        await decompressorExt.deployed();

        return { addr1, addr2, decompressorExt, chainId };
    };

    async function initContractsWithDict () {
        const { addr1, addr2, decompressorExt, chainId } = await
initContracts();

        const dataParts = 20;
        for (let i = 0; i < popularCalldatas.length; i++) {
            if (popularCalldatas[i] === '0x') {
                popularCalldatas.splice(i, 1);
                i--;
            }
            popularCalldatas[i] =
ethers.utils.hexZeroPad(popularCalldatas[i], 32);
        }
        const partIndex = Math.ceil(popularCalldatas.length / dataParts);
        for (let i = 0; i < partIndex; i++) {
            await decompressorExt.setDataArray(i * dataParts + 2,
popularCalldatas.slice(i * dataParts, (i + 1) * dataParts));
        }

        return { addr1, addr2, decompressorExt, chainId };
    }

    describe('Fuzzer', function () {
        it('should decompress random bytes', async function () {
            const { addr1, decompressorExt } = await
loadFixture(initContractsWithDict);
            const calldata =
ethers.utils.hexlify(ethers.utils.randomBytes(1000));
            const result = await compress(calldata, decompressorExt,
addr1.address, popularCalldatas.length);
            const decompressedCalldata = await ethers.provider.call({
                to: decompressorExt.address,
                data:
decompressorExt.interface.encodeFunctionData('decompressed') +
trim0x(result.compressedData),
            });
            expect(ethers.utils.defaultAbiCoder.decode(['bytes'],
decompressedCalldata)).to.deep.eq([calldata]);
        });

        it('should decompress random calldata', async function () {
            const { addr1, decompressorExt } = await
loadFixture(initContractsWithDict);
            for (let i = 0; ; i++) {
                const calldata = generateRandomFunctionCallData();
                console.log('Random function invocation calldata:',
calldata);
                const result = await compress(calldata, decompressorExt,
addr1.address, popularCalldatas.length);
                const decompressedCalldata = await ethers.provider.call({
                    to: decompressorExt.address,
                    data:
decompressorExt.interface.encodeFunctionData('decompressed') +
trim0x(result.compressedData),
                });
                expect(ethers.utils.defaultAbiCoder.decode(['bytes'],
decompressedCalldata)).to.deep.eq([calldata]);
            }
        });
    });
});

function generateRandomFunctionCallData() {
  const functionName = generateRandomString();
  const numParams = Math.floor(Math.random() * 5) + 1; // Random number of
parameters between 1 and 5
  const paramTypes = [];

  for (let i = 0; i < numParams; i++) {
    const paramType = generateRandomType();
    paramTypes.push(paramType);
  }

  const functionSignature = `${functionName}(${paramTypes.join(',')})`;
  const functionSelector = ethers.utils.id(functionSignature).slice(0, 10);

  const args = paramTypes.map((paramType) => {
    return generateRandomArgument(paramType);
  });

  const abiCoder = new ethers.utils.AbiCoder();
  console.log(args);
  const calldata = functionSelector + abiCoder.encode(args.map(arg =>
arg.type), args.map(arg => arg.value)).slice(2);

  return calldata;
}

function generateRandomType() {
  const typeOptions = [
    'address',
    'bool',
    'int',
    'uint',
    'bytes',
    'string',
    'tuple',
    'dynamic'
  ];

  const randomIndex = Math.floor(Math.random() * typeOptions.length);
  const randomType = typeOptions[randomIndex];
  if (randomType === 'dynamic') {
    // Generate random type for dynamic type except for 'tuple'
    while (true) {
      const baseType = generateRandomType();
      if (baseType !== 'tuple') {
        return `${baseType}[]`;
      }
    }
  }

  return randomType;
}

function generateRandomArgument(paramType) {
  switch (paramType) {
    case 'address':
      return {
        type: 'address',
        value:
ethers.utils.getAddress(ethers.utils.hexlify(ethers.utils.randomBytes(20)))
,
      };
    case 'bool':
      return {
        type: 'bool',
        value: Boolean(Math.round(Math.random())),
      };
    case 'int':
        // intRange is an array from 8 up to 128 with a step of 8
        const intRange = ((Math.floor(Math.random() * 16) + 1) * 8);
        const intMax = ethers.BigNumber.from(2).pow(intRange / 2).sub(1);
        const sign = Math.random() < 0.5 ? -1 : 1;
        const intBigNumber =
ethers.BigNumber.from(ethers.utils.randomBytes(16)).mod(intMax).mul(sign);
```

```
        return {
            type: paramType + intRange,
            value: intBigNumber,
        };
    case 'uint':
        const uintRange = (Math.floor(Math.random() * 32) + 1) * 8;
        const uintMax = ethers.BigNumber.from(2).pow(uintRange).sub(1);
        console.log(uintMax);
        const uintBigNumber =
ethers.BigNumber.from(ethers.utils.randomBytes(32)).mod(uintMax);
        return {
            type: paramType + uintRange,
            value: uintBigNumber,
        };
    case 'bytes':
      // length should be greater than 0 but not greater than 32
      const randomLength = Math.floor(Math.random() * 32) + 1;
      return {
        type: `bytes${randomLength}`,
        value:
ethers.utils.hexlify(ethers.utils.randomBytes(randomLength)),
      };
    case 'string':
      const randomString = generateRandomString();
      return {
        type: 'string',
        value: ethers.utils.formatBytes32String(randomString),
      };
    case 'tuple':
        const tupleTypes = [];
        const tupleValues = [];

        generateTuple(tupleTypes, tupleValues);
        return {
          type: `tuple(${tupleTypes.join(',')})`,
          value: tupleValues,
        };
      default:
        if (paramType.endsWith('[]')) {
            const elementType = paramType.slice(0, -2);
            const arraySize = Math.floor(Math.random() * 5) + 1; //
Generate an array of random size
            const arrayValues = [];

            for (let i = 0; i < arraySize; i++) {
                const elementValue = generateRandomArgument(elementType);
// Pass true to indicate dynamic type
                arrayValues.push(elementValue.value);
            }

            return {
                type: `${elementType}[]`,
                value: arrayValues,
            };
        } else {
            throw new Error(`Unknown type: ${paramType}`);
        }
  }

  function generateTuple(currentTupleTypes, currentTupleValues) {
    const tupleSize = Math.floor(Math.random() * 3) + 2;

    for (let i = 0; i < tupleSize; i++) {
      const elementType = generateRandomType();

      if (elementType !== 'tuple') {
        currentTupleTypes.push(elementType);
        const elementValue = generateRandomArgument(elementType);
        currentTupleValues.push(elementValue.value);
      }
    }
  }
}

function generateRandomString() {
  return Math.random().toString(36).substring(2);
}
```

6.2. About us

The Decurity team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained expertise in the blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.