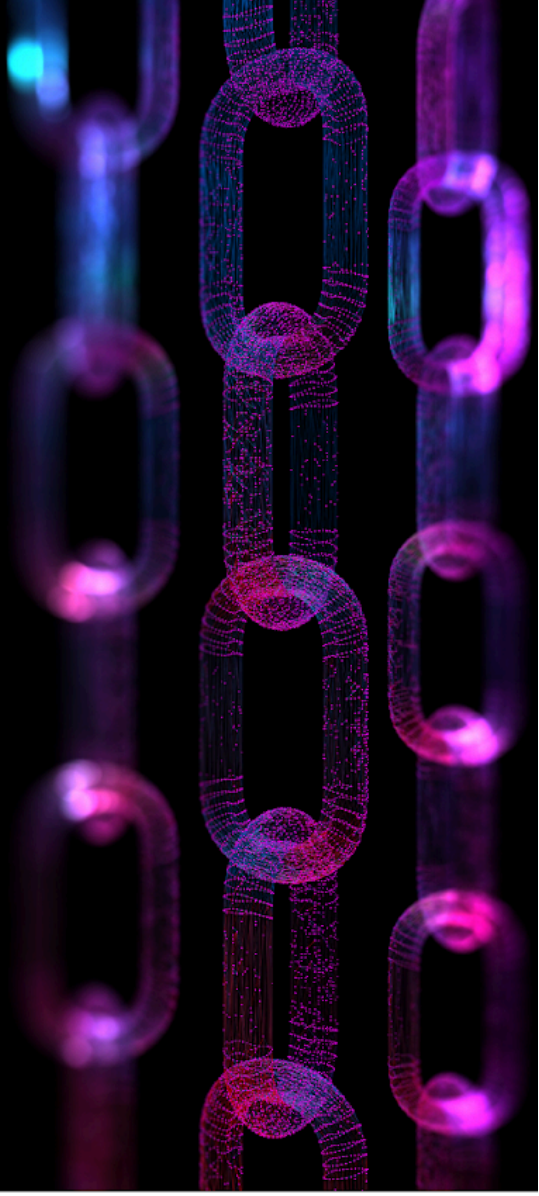


COMPOSABLE SECURITY

REPORT

Smart contract security review for DIVA Technologies AG

Prepared by: Composable Security

Test time period: 2023-03-13 - 2023-04-14

Retest time period: 2023-05-31 - 2023-06-02

Report date: 2023-06-02

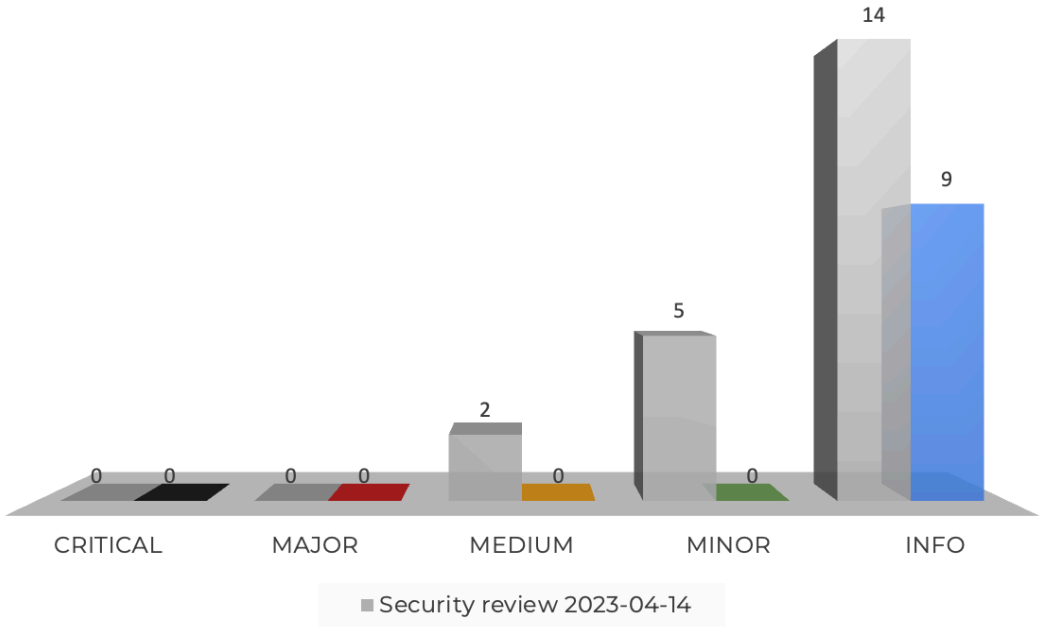
Visit: composable-security.com

Version: 1.2

1. Contents

1. Retest (2023-06-02)

Results diagram



Results

The Composable Security team was involved in a one-time iteration of verification whether the vulnerabilities detected during the tests were removed correctly and no longer appear in the code.

Currently, the status of the identified issues is as follows:

- 2 out of 2 vulnerabilities with a medium impact on risk have been removed from the code.
- 5 out of 5 vulnerabilities with a minor impact on risk have been removed from the code.
- 6 security recommendations were handled as follows:
  - 5 have been implemented,
  - 2 have been partially implemented,
  - 7 have not been implemented.

Scope

Previous security review was carried out 2023-03-13 - 2023-04-14. Verified fixes have been made in the following repository:

DIVA protocol
GitHub repository: https://github.com/divaprotocol/diva-protocol-v1/
CommitID: 54a01507e1412fe88f4efa029bb7e100131052ec

DIVA Tellor oracle adapter
GitHub repository: https://github.com/divaprotocol/oracles/
CommitID: 43acaa4cba8c85145134224dce662572a3d70b72

DIVA token claim contract
GitHub repository: https://github.com/divaprotocol/diva-token-contract/
CommitID: 7111f9c5dc40586e40ce87dd6d84a97068c6735e

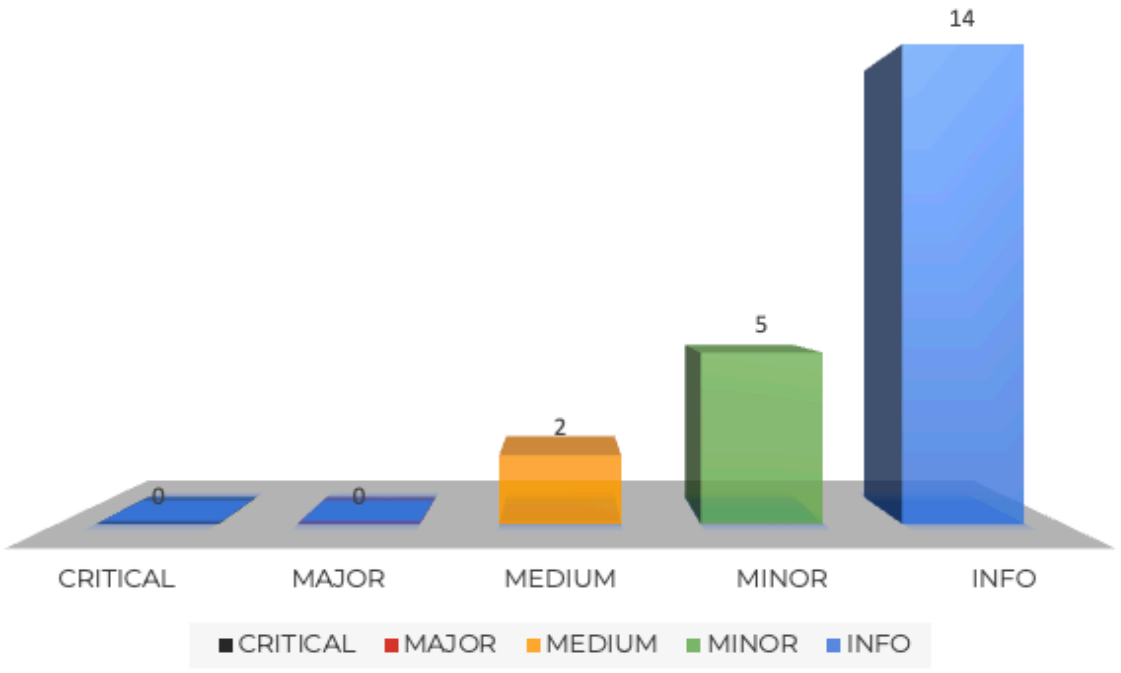
Findings overview

ID	Severity	Vulnerability	Retest 2023-06-02
DIV_0dc20e7_5.1	MEDIUM	Lock of instantly withdrawable funds	FIXED
DIV_0dc20e7_5.2	MEDIUM	Incorrect fallback data provider value	FIXED
DIV_0dc20e7_5.3	MINOR	Incorrect treasury value	FIXED
DIV_0dc20e7_5.4	MINOR	Missing interface	FIXED
DIV_0dc20e7_5.5	MINOR	Unverified position token	FIXED
DIV_0dc20e7_5.6	MINOR	Partially locked token deposits	FIXED
DIV_0dc20e7_5.7	MINOR	Invalid receiver of settlement fee in liquidity removal	FIXED
ID	**Severity**	**Vulnerability**	**Retest 2023-06-02**
DIV_0dc20e7_6.1	INFO	Remove poolId from PoolStorage	IMPLEMENTED
DIV_0dc20e7_6.2	INFO	Follow Checks-Effects-Interactions pattern	PARTIALLY IMPLEMENTED
DIV_0dc20e7_6.3	INFO	Improve code clarity	PARTIALLY IMPLEMENTED
DIV_0dc20e7_6.4	INFO	Use proper error for non-existing pool	IMPLEMENTED
DIV_0dc20e7_6.5	INFO	Add incentive for the default settlement	NOT IMPLEMENTED
DIV_0dc20e7_6.6	INFO	Optimize gas consumption by removing redundant checks	IMPLEMENTED
DIV_0dc20e7_6.7	INFO	Avoid zero value transfers initiated by the protocol	NOT IMPLEMENTED
DIV_0dc20e7_6.8	INFO	Detect duplicates in claimers' addresses	IMPLEMENTED
DIV_0dc20e7_6.9	INFO	Consider adding white hat hacks policy	NOT IMPLEMENTED

DIV_0dc20e7_6.10	INFO	Remove payable mutability from withdraw function	NOT IMPLEMENTED
DIV_0dc20e7_6.11	INFO	Consider adding popups for front-end application to warn users	NOT IMPLEMENTED
DIV_0dc20e7_6.12	INFO	Consider extending the effect of the pauseReturnCollateral function	NOT IMPLEMENTED
DIV_0dc20e7_6.13	INFO	Protect withdrawing all tokens before setting up trigger	NOT IMPLEMENTED
DIV_0dc20e7_6.14	INFO	Use the same version of Solidity in all smart contracts (latest stable)	IMPLEMENTED

2. Executive summary

2.1. Audit results diagram (2023-04-20)



2.2. Audit results

The *DIVA Technologies AG* engaged Composable Security to review security of *DIVA protocol* smart contracts. Composable Security conducted this assessment over 5 person-weeks with 2 engineers.

The scope of the tests included selected contracts from the following repositories.

DIVA protocol
GitHub repository: https://github.com/divaprotocol/diva-contracts/
CommitID: *0dc20e7ada71b45518ebb85cacdfc8cf05940a38*

DIVA Tellor oracle adapter
GitHub repository: https://github.com/divaprotocol/oracles/
CommitID: *3dbd56e7e7bba24bbd7cf6d75146157a4921cc97*

DIVA token claim contract
GitHub repository: https://github.com/divaprotocol/diva-token-contract/
CommitID: *30448bbd014b037ee9245c32233f79d16689c02c*

Note regarding GitHub repositories:
The divaprotocol/diva-contracts repository has been transferred to a new repository divaprotocol/diva-protocol-v1.
We want to confirm that the source code of all contracts covered during the audit is the same on the following commit ID: *72bcd26c963554799b2860f929108df33409b45b*

Audit findings:

- Neither critical nor high risk impact vulnerabilities were identified.
- 2 vulnerabilities with a medium impact on risk were identified. Their potential consequences are:
    - The funds transferred to *DIVADevelopmentFund* as instantly withdrawable are locked. (5.1)
    - The settlement fee is transferred to an incorrect fallback data provider. (5.2)
- 5 vulnerabilities with a low impact on risk were identified.
- 14 recommendations have been proposed that can improve overall security and help implement best practice.
- The protocol accepts any type of collateral tokens, data providers and permissioned NFTs therefore it is important to inform the users about the risks on the frontend application side (see 6.11).

Note: The team was involved in the audit and communication was very smooth. We would like to highlight that the code is well thought out and the documentation that comes with it is of high quality. All functions contain the necessary descriptions, and the key flows are clearly described. It also includes the warnings and risks descriptions associated with using the system in a transparent manner.

Composable Security recommends that DIVA Technologies AG complete the following:

- Address all reported vulnerabilities.
- Extend unit tests with scenarios that cover detected vulnerabilities where possible.

- Consider whether the detected vulnerabilities may exist in other places (or ongoing projects) that have not been detected during engagement.
- Consider implementation of proposed recommendations.
- Consider implementation of proposed long term best practices

3. Project details

3.1. Projects goal

The Composable Security team focused during this audit on the following:

- Perform a tailored threat analysis.
- Ensure that smart contract code is written according to security best practices.
- Identify security issues and potential threats both for *DIVA protocol* and their users.

The secondary goal is to improve code clarity and optimize code where possible.

3.2. Agreed scope of tests

The subjects of the test were selected contracts from the *DIVA protocol* repositories.

DIVA protocol
GitHub repository: https://github.com/divaprotocol/diva-contracts/
CommitID: *0dc20e7ada71b45518ebb85cacdfc8cf05940a38*

Files in scope:

```
.
├── DIVADevelopmentFund.sol
├── DIVAOwnershipMain.sol
├── DIVAOwnershipSecondary.sol
├── Diamond.sol
├── PermissionedPositionToken.sol
├── PositionToken.sol
├── PositionTokenFactory.sol
├── UsingTellor.sol
├── facets
│   ├── ClaimFacet.sol
│   ├── DiamondCutFacet.sol
│   ├── DiamondLoupeFacet.sol
│   ├── EIP712AddFacet.sol
│   ├── EIP712CancelFacet.sol
│   ├── EIP712CreateFacet.sol
│   ├── EIP712RemoveFacet.sol
│   ├── GetterFacet.sol
│   ├── GovernanceFacet.sol
│   ├── LiquidityFacet.sol
│   ├── PoolFacet.sol
│   └── SettlementFacet.sol
└── interfaces
    ├── IClaim.sol
    ├── IDIVADevelopmentFund.sol
    ├── IDIVAOwnership.sol
    ├── IDiamondCut.sol
    ├── IDiamondLoupe.sol
    ├── IEIP712Add.sol
    ├── IEIP712Cancel.sol
    ├── IEIP712Create.sol
    ├── IEIP712Remove.sol
    ├── IERC165.sol
    ├── IGetter.sol
    ├── IGovernance.sol
    ├── ILiquidity.sol
    ├── IPermissionedPositionToken.sol
    ├── IPool.sol
    ├── IPositionToken.sol
```

```
│   ├── IPositionTokenFactory.sol
│   └── ISettlement.sol
└── libraries
    ├── LibDIVA.sol
    ├── LibDIVAStorage.sol
    ├── LibDiamond.sol
    ├── LibDiamondStorage.sol
    ├── LibEIP712.sol
    ├── LibEIP712Storage.sol
    ├── LibOwnership.sol
    └── SafeDecimalMath.sol
```

DIVA Tellor oracle adapter

GitHub repository: https://github.com/divaprotocol/oracles/

CommitID: *3dbd56e7e7bba24bbd7cf6d75146157a4921cc97*

Files in scope:

```
.
├── DIVAOracleTellor.sol
├── UsingTellor.sol
├── interfaces
│   ├── IDIVAOracleTellor.sol
│   └── IDIVA.sol
└── libraries
    └── SafeDecimalMath.sol
```

DIVA token claim contract

GitHub repository: https://github.com/divaprotocol/diva-token-contract/

CommitID: *30448bbd014b037ee9245c32233f79d16689c02c*

Files in scope:

```
.
└── ClaimDIVALinearVesting.sol
```

Documentation:

https://github.com/divaprotocol/diva-contracts/blob/main/DOCUMENTATION.md

https://github.com/divaprotocol/oracles/blob/main/docs/Tellor.md

https://github.com/divaprotocol/diva-token-contract/blob/main/README.md#claim-contract-with-vesting-logic

3.3. Threat analysis

This section summarizes the potential threats that were identified during initial threat modeling performed before the audit. The tests were focused, but not limited to, finding security issues that could be exploited to achieve these threats.

Before starting the tests, a high-level diagram was created containing the key business flows of the *DIVA protocol*.

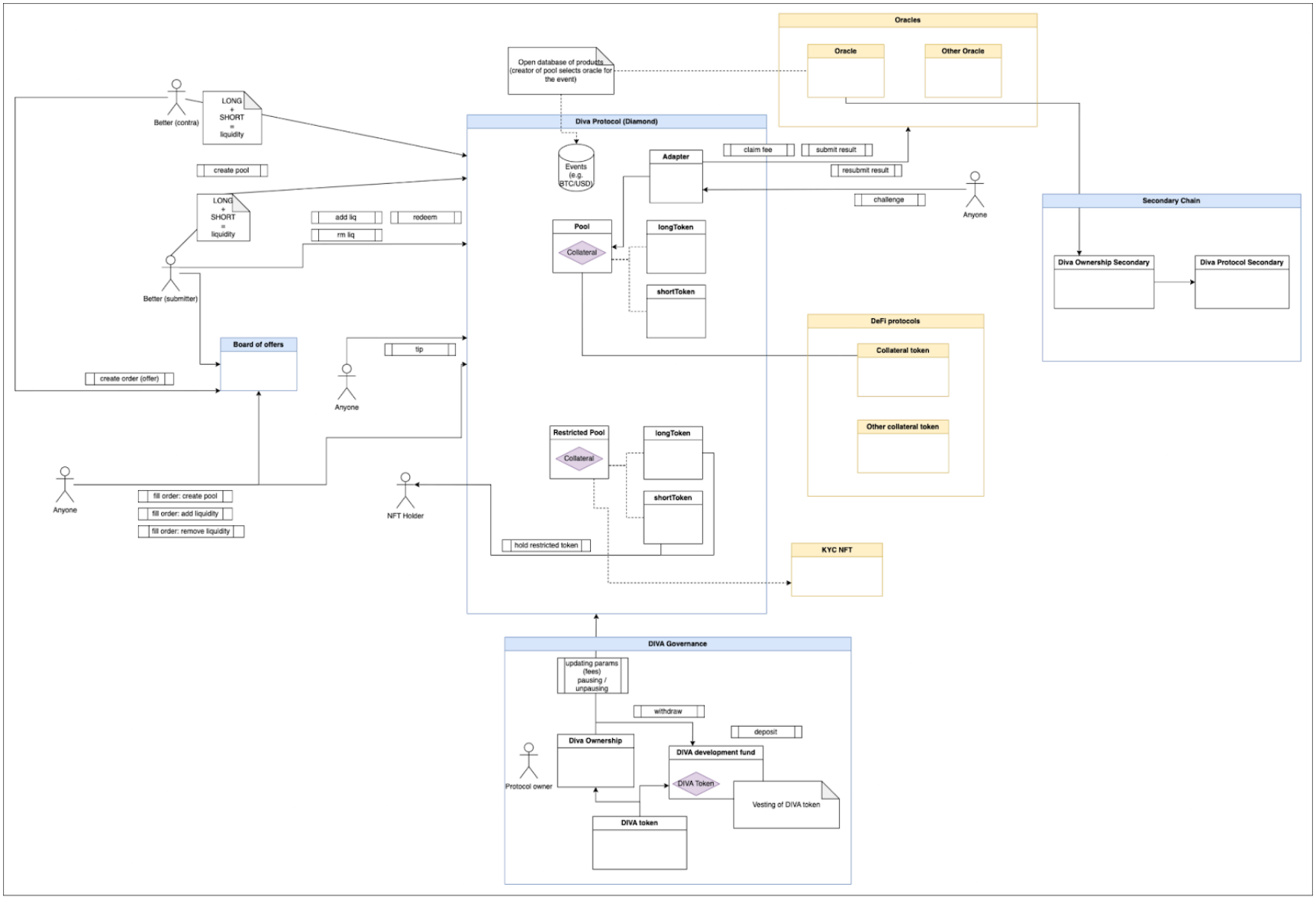
The most important mechanics include:

- creation of contingents pools,
- settlement process.

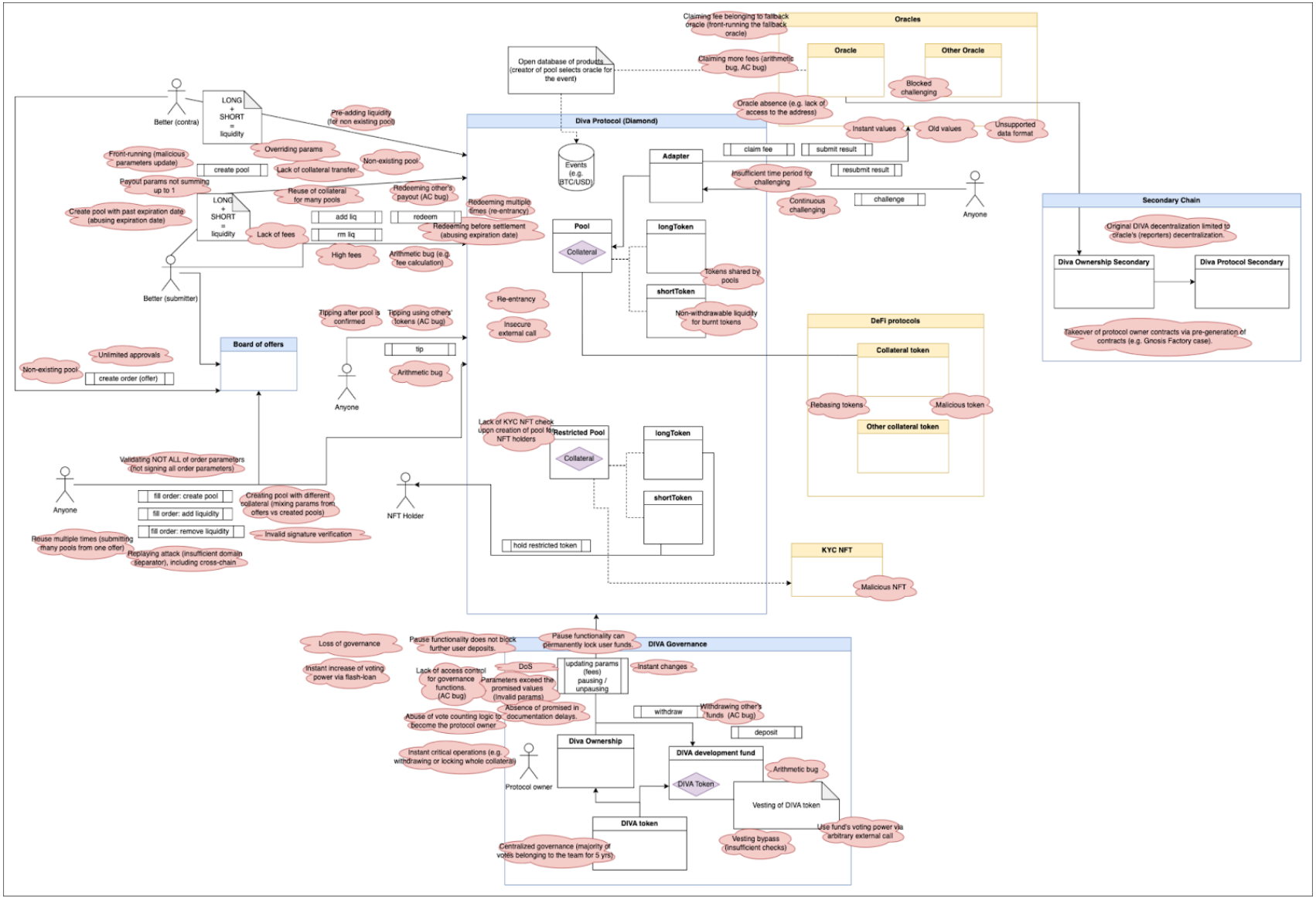
Potential attackers goals:

- Steal collateral from the pools.
- Exceed the established rules to get bigger rewards, fees.
- Exceed the established rules to unleash tokens earlier.
- Steal tokens from Development Fund.
- DoS protocol.
- Give the protocol a bad reputation.
- Steal protocol fees.
- Take control of the protocol.
- Don't lose collateral when you lose the bet.
- Lock users' funds in the contract.
- Self-management without obtaining the required threshold.
- Bypass regulations (KYC).

The diagram below represents the high-level business flow for key operations.



The diagram below contains **potential threats** identified. For the avoidance of doubt, **these DO NOT represent issues discovered during the audit**, but rather form the basis of the potential vulnerabilities to watch out for during the audit and the security review.



Potential scenarios to achieve the indicated attacker's goals:

- Pre-adding liquidity for non existing pools.
- Lack of collateral transfer.
- Influence or bypass the business logic of the system.
- Take advantage of arithmetic errors.
- Privilege escalation through incorrect access control to functions or badly written modifiers.
- Existence of known vulnerabilities (e.g., front-running, re-entrancy).
- Design issues.
- Excessive power, too much in relation to the declared one.
- Unintentional loss of the ability to govern the system.
- Poor security against taking over the managing account.
- Private key compromise, rug-pull.
- Withdrawal of more funds than expected.
- Oracle manipulation.
- Abuse of incorrect arithmetic operations and calculations.

3.4. Testing methodology

Smart contract security review was performed using the following methods:

- Q&A sessions with the *DIVA Technologies AG* development team to thoroughly understand intentions and assumptions of the project.
- Initial threat modeling to identify key areas and focus on covering the most relevant scenarios based on real threats.
- Automatic tests using *slither*.
- Custom scripts (e.g. unit tests) to verify scenarios from initial threat modeling.
- **Manual review of the code.**

3.5. Disclaimer

Smart contract security review **IS NOT A SECURITY WARRANTY**.

During the tests, the Composable Security team makes every effort to detect any occurring problems and help to address them. However, it is not allowed to treat the report as a security certificate and assume that the project does not contain any vulnerabilities. Securing smart contract platforms is a multi-stage process, starting from threat modeling, through development based on best practices, security reviews and formal verification, ending with constant monitoring and incident response.

Therefore, we encourage the implementation of security mechanisms at all stages of development and maintenance.

4. Findings overview

ID	Severity	Vulnerability
DIV_0dc20e7_5.1	MEDIUM	Lock of instantly withdrawable funds
DIV_0dc20e7_5.2	MEDIUM	Incorrect fallback data provider value
DIV_0dc20e7_5.3	MINOR	Incorrect treasury value
DIV_0dc20e7_5.4	MINOR	Missing interface
DIV_0dc20e7_5.5	MINOR	Unverified position token
DIV_0dc20e7_5.6	MINOR	Partially locked token deposits
DIV_0dc20e7_5.7	MINOR	Invalid receiver of settlement fee in liquidity removal
ID	**Severity**	**Recommendation**
DIV_0dc20e7_6.1	INFO	Remove poolId from PoolStorage
DIV_0dc20e7_6.2	INFO	Follow Checks-Effects-Interactions pattern
DIV_0dc20e7_6.3	INFO	Improve code clarity
DIV_0dc20e7_6.4	INFO	Use proper error for non-existing pool
DIV_0dc20e7_6.5	INFO	Add incentive for the default settlement
DIV_0dc20e7_6.6	INFO	Optimize gas consumption by removing redundant checks
DIV_0dc20e7_6.7	INFO	Avoid zero value transfers initiated by the protocol
DIV_0dc20e7_6.8	INFO	Detect duplicates in claimers' addresses
DIV_0dc20e7_6.9	INFO	Consider adding white hat hacks policy
DIV_0dc20e7_6.10	INFO	Remove payable mutability from withdraw function
DIV_0dc20e7_6.11	INFO	Consider adding popups for front-end application to warn users
DIV_0dc20e7_6.12	INFO	Consider extending the effect of the pauseReturnCollateral function
DIV_0dc20e7_6.13	INFO	Protect withdrawing all tokens before setting up trigger
DIV_0dc20e7_6.14	INFO	Use the same version of Solidity in all smart contracts (latest stable)

5. Vulnerabilities

5.1. Lock of instantly withdrawable funds

Status 2023-06-02	FIXED

The vulnerability has been removed as recommended. The *_isValidReleasePeriod* function has been added in both deposit functions.

It now verify *_releasePeriodInSeconds* as follows:

```
(...)
function _isValidReleasePeriod(uint256 _releasePeriodInSeconds) private pure returns
(bool) {
          return (_releasePeriodInSeconds != 0 && _releasePeriodInSeconds <= 30*365
days);
}
(...)
```

Severity

MEDIUM

Affected smart contracts

DIVADevelopmentFund.sol

Description

The *deposit* functions allow you to support the team by transferring tokens. The person sending the funds can define how quickly they will be released by the *_releasePeriodInSeconds* parameter.

In case the parameter is set to *0* (i.e. we transfer funds and want to allow *DIVAOwner* to use them immediately, without waiting for the time to pass). The funds will be properly deposited, but it will not be possible to withdraw them.

There are two functions allowing for withdrawals:

1. The *withdraw* (for deposit through custom functions) will not allow you to withdraw funds due to a check that will never be fulfilled (*DIVADevelopmentFund.sol#L88*):

```
74      function withdraw(address _token, uint256[] calldata _indices)
75        external
76        payable
77        override
78        nonReentrant
79        onlyDIVAOwner
80    {
81        uint256 _claimableAmount;
82        uint256 _len = _indices.length;
```

```
83          for (uint256 _i = 0; _i < _len; ) {
84              Deposit storage _deposit = _deposits[_indices[_i]];
85              if (_deposit.token != _token) {
86                  revert DifferentTokens();
87              }
88              if (_deposit.lastClaimedAt < _deposit.endTime) {
89                  _claimableAmount += _claimableAmountForDeposit(_deposit);
90                  _deposit.lastClaimedAt = block.timestamp;
91              }
92
93              unchecked {
94                  ++_i;
95              }
96          }
97
98          _tokenToUnclaimedDepositAmount[_token] -= _claimableAmount;
99
100         if (_token == address(0)) {
101             (bool success, ) = msg.sender.call{value: _claimableAmount}("");
102             require(success, "Failed to send native asset");
103         } else {
104             IERC20(_token).safeTransfer(msg.sender, _claimableAmount);
105         }
106
107         emit Withdrawn(msg.sender, _token, _claimableAmount);
108     }
```

2. The *withdrawDirectDeposit* (for direct deposits) will not allow to withdraw funds due to excluding tokens deposited through custom functions through *_tokenToUncalimedDepositAmount* parameter (DIVADevelopmentFund.sol#L126-128).

```
110 function withdrawDirectDeposit(address _token)
111         external
112         payable
113         override
114         nonReentrant
115         onlyDIVAOwner
116     {
117         uint256 _claimableAmount;
118         if (_token == address(0)) {
119             _claimableAmount =
120                 address(this).balance -
121                 _tokenToUnclaimedDepositAmount[_token];
122             (bool success, ) = msg.sender.call{value: _claimableAmount}("");
123             require(success, "Failed to send native asset");
124         } else {
125             IERC20 _depositTokenInstance = IERC20(_token);
126             _claimableAmount =
127                 _depositTokenInstance.balanceOf(address(this)) -
128                 _tokenToUnclaimedDepositAmount[_token];
129             IERC20(_token).safeTransfer(msg.sender, _claimableAmount);
130         }
131
132         emit Withdrawn(msg.sender, _token, _claimableAmount);
133     }
134
```

Vulnerable scenario

The scenario in which the funds will be blocked may look as follows:

- A user deposits 10,000 USDC by calling a *deposit* function to support the development of the project and wants his funds to be used immediately, so he sets the release period to *0*.
- *DIVAOwner* tries to withdraw the transferred funds, but the transaction reverts due to the lack of fulfillment of the condition ([*DIVADevelopmentFund.sol#L88*]).

Result: The funds transferred to *DIVADevelopmentFund* as instantly withdrawable are locked.

Recommendation
Based on the conversation with the team, it was determined that from a business point of view, there is no need for deposits to be immediately withdrawable. Therefore, the case where *_releasePeriodInSeconds = 0* can be safely excluded from use. In this case, you should add appropriate input parameter validation to make sure that *_releasePeriodInSeconds* is higher than 0 in both deposit functions.

References

SCSVS G4: Business logic
https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x100-General/0x104-G4-Business-Logic.md

5.2. Incorrect fallback data provider value

Status 2023-06-02	FIXED

The vulnerability has been removed.

The *_confirmFinalReferenceValue* function now uses *msg.sender* to set the current data provider:

```
(...)
// If within the fallback period (the case when the data provider
// failed to submit a value). Note that `block.timestamp > submissionEndTime`
// at this point.

(...)
   else if (
     block.timestamp<=submissionEndTime+_settlementPeriods.fallbackSubmissionPeriod
            ) {
               // Check that the `msg.sender` is the fallback data provider
               if (msg.sender != LibDIVA._getCurrentFallbackDataProvider(_gs))
                  revert NotFallbackDataProvider();

                _confirmFinalReferenceValue(
                    _poolId,
                    _pool,
                    _treasury,
                    msg.sender,
                    _finalReferenceValue,
                    _gs
```

```
                    );
                }
(...)
```

Severity

MEDIUM

Affected smart contracts

SettlementFacet.sol

Description

The *SettlementFacet* contract contains a *_setFinalReferenceValue* function that is responsible for setting the final value for the contingent pool and update fees. One of the cases is when the data provider is absent and the fallback data provider sets this value (in a defined timeframe).

However, during the update fallback data provider period (60 days), the settlement fee is transferred to the new fallback data provider instead of the current one who keeps their role until the period is finished.

Vulnerable scenario

- The owner updates the fallback data provider.
- The contingent pool becomes expired and the data provider does not submit the value.
- The current fallback data provider submits the value.
- The settlement fee is transferred to the new fallback data provider.

Result: The settlement fee is transferred to an incorrect fallback data provider. It also takes away the incentive from the fallback data providers to submit values.

Recommendation
Use the result of *_getCurrentFallbackDataProvider* function as the settlement fee recipient.

References

SCSVS G4: Business logic
https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x100-General/0x104-G4-Business-Logic.md

5.3. Incorrect treasury value

Status 2023-06-02	FIXED

The vulnerability has been removed.

The protocol fee reserved for DIVA treasury handled by *_allocateFeeClaim* function, now uses *getCurrentTreasury()* to set appropriate treasury:

```
(...)
// Allocate protocol fee to DIVA treasury. Fee is held within this
// contract and can be claimed via `claimFee` function.
// `collateralBalance` is reduced inside `_allocateFeeClaim`.

        _allocateFeeClaim(
            _removeLiquidityParams.poolId,
            _pool,
            _getCurrentTreasury(gs),
            _protocolFee
        );
(...)
```

Severity

MINOR

Affected smart contracts

LibDIVA.sol

Description

When a user removes the liquidity from the contingent pool, the *_removeLiquidityLib* function allocates the protocol (*LibDIVA.sol#L780*) and settlement (*LibDIVA.sol#L789*) fees.

However, during the update treasury period (2 days), the protocol fee is transferred to the new treasury instead of the current one, which should remain for 2 days after the call to update function.

Note: The severity of this finding has been lowered as the invalid value is kept for 2 days (instead of 60 days in case of fallback data provider) and it does not remove any incentives.

Vulnerable scenario

- The owner updates the treasury.
- A user removes the liquidity from the contingent pool.
- The protocol fee is transferred to the new treasury, which is a candidate, and not the current treasury.

Result: Transfer of protocol fee to incorrect treasury, according to the documentation.

Recommendation
Use the result of *_getCurrentTreasury* function as the protocol fee recipient.

References

SCSVS G4: Business logic
https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x100-General/0x104-G4-Business-Logic.md

5.4. Missing interface

Status 2023-06-02	FIXED

The vulnerability has been removed as recommended.

The custom DIVA protocol interfaces have been removed from the ERC165 lookup table:

```
(...)
        // Adding ERC165 data
        ds.supportedInterfaces[type(IDiamondLoupe).interfaceId] = true;
        ds.supportedInterfaces[type(IERC165).interfaceId] = true;
(...)
```

Severity

MINOR

Affected smart contracts

Diamond.sol

Description

The *Diamond* contract uses ERC-165: Standard Interface Detection and supports the interfaces using the *supportedInterfaces* mapping (*LibDiamondStorage.sol#L30*). All supported interfaces (including IERC165 itself, Diamond's interfaces and all DIVA protocol interfaces) are added to the mentioned mapping in Diamond's constructor (*Diamond.sol#L154*). However, the *ITip* interface is missing.

Result:
In case of integrations (or frontend applications connecting) with *DIVA* protocol, if they use ERC165 to lookup for specific supported interfaces, it would not be possible to lookup the *ITip* interface.

However, as stated in the recommendation section, it is unlikely that other protocols would lookup custom *DIVA protocol* interfaces.

Recommendation
The general recommendation is to remove the custom *DIVA protocol* interfaces from the ERC165 lookup table as the other protocols would unlikely lookup custom interfaces. In case the team plans to make those interfaces a common ones (used by multiple protocols), we recommend adding the *ITip* interfaces to the ERC165 lookup table.

References

SCSVS I1: Basic
https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x300-Integrations/0x301-I1-Basic.md

5.5. Unverified position token

Status 2023-06-02	FIXED

The vulnerability has been removed as recommended.

The function *getPoolParametersByAddress* now verifies the passed position token if it is one of the pool's position tokens:

```
(...)
    function getPoolParametersByAddress(address _positionToken)
        external
        view
        override
        returns (LibDIVAStorage.Pool memory)
    {
        // Read the `poolId` from the `PositionToken` contract
        PositionToken positionToken = PositionToken(_positionToken);
        bytes32 _poolId = positionToken.poolId();

        // Load pool information
   LibDIVAStorage.Pool storage _pool = LibDIVAStorage._poolStorage().pools[_poolId];
        // Return pool information only if the provided position token address is
valid.
        // Otherwise, return the default struct.
        if (_pool.shortToken == _positionToken || _pool.longToken == _positionToken)
{
            return _pool;
        }
        LibDIVAStorage.Pool memory _zeroPool;
        return _zeroPool;
    }
(...)
```

Severity

MINOR

Affected smart contracts

GetterFacet.sol

Description

The *getPoolParametersByAddress* function (GetterFacet.sol#L26) from *GetterFacet* contract is used to get the pool parameters for a given position token address. It takes the pool identifier from the passed address and asks *DIVA protocol* about the parameters.

However, the function does not check whether the passed position token address is the legitimate token for a given pool identifier. The malicious user could provide a fake position token contract.

Note: The severity of this finding has been lowered as the testing team was not able to verify how the mentioned function is used by applications that integrate with *DIVA protocol*.

Attack scenario

The following scenario assumes that the application that integrates with *DIVA* protocol uses the passed position token contract for further operations:

- The attacker deploys a fake position token with a legitimate pool identifier.
- The victim asks the protocol, indirectly by some vulnerable application, about the pool parameters for fake token and retrieves the legitimate pool parameters.
- The attacker continues operations in the vulnerable application using the fake position token and pool parameters (e.g. sells fake position token based on legitimate parameters, including collateral value).

Result of the attack:
The example scenario could lead to trading fake position tokens. However, the result of the attack depends on how the applications that integrate with *DIVA* protocol and use *getPoolParametersByAddress* function handle the passed argument.

Recommendation
Add a requirement that one of the pool's position tokens is the passed position token.

References

SCSVS I2: Token
https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x300-Integrations/0x302-I2-Token.md

5.6. Partially locked token deposits

Status 2023-06-02	FIXED

The vulnerability has been removed.

The protocol does not accept fee-on-transfer tokens and reverts when such are used. There are similar checks added in functions that transfer the tokens. They check whether the actual amount transferred is equal to the expected amount and if not, the transaction reverts (see the example below).

In the described case of the function *deposit*, now it verifies the balances of the development fund before and after the *safeTransferFrom* call to reflect the transferred amount correctly:

```
(...)
    function deposit(
        address _token,
        uint256 _amount,
        uint256 _releasePeriodInSeconds
    ) external override nonReentrant {
        if (!_isValidReleasePeriod(_releasePeriodInSeconds)) {
            revert InvalidReleasePeriod();
        }
        uint256 _depositIndex = _addNewDeposit(
            _token,
            _amount,
            _releasePeriodInSeconds
        );

        IERC20 _tokenInstance = IERC20(_token);

        // Transfer token from user to `this`. Revert if a fee is applied
        // during transfer.
        uint256 _before = _tokenInstance.balanceOf(address(this));
        _tokenInstance.safeTransferFrom(msg.sender, address(this), _amount);
        uint256 _after = _tokenInstance.balanceOf(address(this));

        if (_after - _before != _amount) {
            revert FeeTokensNotSupported();
        }

        emit Deposited(msg.sender, _depositIndex);
    }
(...)
```

Severity

MINOR

Affected smart contracts

DIVADevelopmentFund.sol

Description

The *deposit* function in the *DIVADevelopmentFund* contract (DIVADevelopmentFund.sol#L58) is used to deposit tokens that are later released linearly. The deposit struct saves the amount of tokens passed in the argument and later transfers the same amount using the *transferFrom* function called on the token contract. In case of tokens, such as rebasing or fee-on-transfer tokens, the partial amount of deposited tokens can be locked.

Note: The severity of this finding has been lowered because one of its requirements is to receive fee-on-transfer, rebasing or other, uncommon token by the fund contract. Additionally, in case of full amount locked on the contract, to unlock the whole amount of tokens, the owner can deposit the missing amount (e.g. the fee amount).

Vulnerable scenario

The following scenario locks the partial amount of tokens:

1. The depositor deposits 100 tokens (supporting fee on transfer) to the fund.
2. The fund contract saves the amount equal to 100 and transfers 100 tokens from the sender to the fund contract, but 1 token is transferred to the fee recipient (leaving 99 tokens on the fund contract).
3. The owner waits until the 50% of tokens is claimable and withdraws it. The fund contract transfers 50 tokens of the whole amount and decreases the amount value to the remaining 50 tokens.
4. When the deposit period is finished, the owner tries to withdraw the rest of the tokens, but the transaction reverts because the amount to be transferred is 50 tokens while the balance of the fund contract is 49 tokens.

If the owner omits step 3 and wants to withdraw the full amount in step 4, the whole amount is locked and cannot be withdrawn without sending missing tokens to the fund contract.

Similarly, for the rebase tokens, if the total supply changes, the transferable amount can also change and become insufficient.

Result:
The partial amount of tokens can be locked and will have to be transferred to the development fund in order to unlock it.

Recommendation

- Either check the balances of the development fund before and after the *transferFrom* call to reflect the transferred amount correctly; or
- Communicate that the fund accepts only a selected list of tokens (this can also be implemented using a whitelist, but will also increase the cost of deposit).

References

SCSVS I2: Token
https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x300-Integrations/0x302-I2-Token.md

5.7. Invalid receiver of settlement fee in liquidity removal

Status 2023-06-02	FIXED

The vulnerability has been removed as recommended.

The reserve settlement fee for data providers is now handled by *_reserveFeeClaim* function:

```
(...)
// Reserve settlement fee for data provider which is not known at this stage.
// Fee will be allocated to actual data provider following final value
// confirmation and afterwards can be claimed via the `claimFee` function.

        _reserveFeeClaim(
            _removeLiquidityParams.poolId,
            _pool,
            _settlementFee
        );
(...)
```

This function increases fee claim reserve for the actual data provider:

```
(...)
    function _reserveFeeClaim(
        bytes32 _poolId,
        LibDIVAStorage.Pool storage _pool,
        uint256 _feeAmount
    ) internal {
        // Check that fee amount to be reserved doesn't exceed the pool's
        // current `collateralBalance`. This check should never trigger, but
        // kept for safety.
        if (_feeAmount > _pool.collateralBalance)
            revert FeeAmountExceedsPoolCollateralBalance();

        // Reduce `collateralBalance` in pool parameters and increase
        // fee claim reserve
        _pool.collateralBalance -= _feeAmount;
        LibDIVAStorage._feeClaimStorage()
            .poolIdToReservedClaim[_poolId] += _feeAmount;

        // Log poolId and fee amount
        emit FeeClaimReserved(_poolId, _feeAmount);
    }
(...)
```

Severity

MINOR

Affected smart contracts

LibDIVA.sol

Description

The liquidity provider is allowed to remove liquidity at any time (if they have enough of both long and short position tokens) before the pool's status becomes confirmed. Removing liquidity is subject to the protocol and settlement fee. The latter one is allocated to the data provider that has been selected for the pool (LibDIVA.sol#L792).

However, if the data provider does not report the outcoming during the reporting window and the fallback data provider or the default value (inflection) is used to settle the pool, the settlement fee is not transferred to the factual data provider.

Vulnerable scenario

- The user A creates a pool and selects DP as data provider.
- The user B adds liquidity.
- The user B removes liquidity and pays protocol fee (allocated to treasury) and settlement fee allocated to DP.
- After the pool is expired, the DP is absent, the fallback data provider submits the final reference value and gets the current settlement fee (calculated for the current collateral).
- The DP withdraws the fee that was allocated to them on liquidity removal while should be allocated to the fallback data provider.

Result:
The settlement fee taken on the liquidity withdrawal is always allocated to the selected data provider, even if they are not the factual data provider and someone else submits the final reference value.

Recommendation
Save the fee intended for the data provider in a separate variable and allocate it when the final reference value is set.

References

SCSVS G4: Business logic
https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x100-General/0x104-G4-Business-Logic.md

6. Recommendations

6.1. Remove *poolId* from *PoolStorage*

Status 2023-06-02	IMPLEMENTED

The *poolId* is no longer a *uint256*, so the recommendation is no longer applicable.

Currently pools are identified as the hash of create pool parameters, *msg.sender* and *nonce*.

```
(...)
// IMPORTANT: The hash calculation in `LibDIVA._getPoolId()` assumes
// that the `nonce` variable is stored at slot 0 inside the `PoolStorage` struct

    struct PoolStorage {
        uint256 nonce;
        mapping(bytes32 => Pool) pools; // poolId => Pool struct
        address positionTokenFactory;
    }
(...)
```

Severity

INFO

Description

The *poolId* field in *PoolStorage* (LibDIVAStorage.sol#L86) is used to store the identifier of the latest pool. It is not necessary to store it in a separate value as it can easily be obtained with the length of *pools* variable from the same struct.

Recommendation
Remove the *poolId* variable and use the length of *pools* variable instead.

References

SCSVS G11: Code clarity
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x111-G11-Code-Clarity.md

6.2. Follow Checks-Effects-Interactions pattern

Status 2023-06-02	PARTIALLY IMPLEMENTED

The recommendation was partially implemented. The CEI pattern was introduced in *_addLiquidityLib* function.

The *stake* function is still not following the recommended pattern, but the token it uses is the DIVA token that does not pose the risk.

Additionally, the rest of places where the CEI pattern is not followed have been discussed and no security risk arises from them. Moreover, all functions that modify the state are protected with the *nonReentrant* modifier.

Severity

INFO

Description:

The protocol uses the *nonReentrant* modifier to protect from re-entrancy and in most of the functions follow the Checks-Effects-Interactions pattern. However, there are functions that violate this best practice:

- Function *stake* - DIVAOwnershipMain.sol#L71,
- Function *_addLiquidityLib* - LibDIVA.sol#L631.

Recommendation
Make sure that interactions are implemented after checks and effects.

References:

SCSVS G6: Communications
https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x100-General/0x106-G6-Communications.md

6.3. Improve code clarity

Status 2023-06-02	PARTIALLY IMPLEMENTED

The recommendation was partially implemented.

In the case of *MAX_FEE* and *MIN_FEE* scientific notation was used. However, the team decided not to introduce immutable variables for them.

The team decided not to add short descriptions at the beginning of the selected smart

contracts.

Severity

INFO

Description

Due to the composable nature of the smart contract ecosystem, it is recommended to keep the code clear and highly understandable. No leftovers or unused code snippets should be left in the code. Describe your assumptions, reuse code from trustworthy sources when possible and do not complicate the implementation.

This will not only allow you to avoid unwanted issues, but also maximize the effectiveness of audits and peer reviews.

Recommendation

- Update the description of *lastClaimedAt* in documentation to properly reflect the behavior of the code.
    - The *lastClaimedAT* variable indeed represents timestamp in seconds since epoch when user last claimed deposit at, but AFTER first claim. Before the first claim it is timestamp in seconds since the epoch when the user can start claiming the deposit.
- Conform to Solidity naming convention
    - Implement immutable variables for MAX_FEE and MIN_FEE (use scientific notation to increase readability).
        - GovernanceFacet.sol#L408-409
    - Use UPPER_CASE_WITH_UNDERSCORES for constant variables
        - DIVAOwnershipMain.sol#L45, DIVAOwnershipMain.sol#L58-61
        - DIVAOwnershipSecondary.sol#L37-40
        - DIVADevelopmentFund.sol#L14
        - PositionTokenFactory.sol#L16-17
- Do not repeat code
    - Use `LibDiamondStorage.DiamondStorage storage ds = LibDiamondStorage._diamondStorage()` (Diamond.sol#L162)
- Remove unused function (*positionTokenImplementation*) or add a getter function for *_permissionedPositionTokenImplementation* variable if you want to let users know the address of the implementation contract.
    - The function *positionTokenImplementation* in *PositionTokenFactory* is never used and there is no similar function for the *_permissionedPositionTokenImplementation* variable (PositionTokenFactory.sol#L60).

- Consider adding short descriptions at the beginning of the following smart contracts
  - *PermissionedPositionToken*
  - *DIVADevelopmentFund*
  - *ClaimFacet*
  - *EIP712AddFacet*
  - *EIP712CancelFacet*
  - *EIP712CreateFacet*
  - *EIP712RemoveFacet*
  - *GetterFacet*
  - *GovernanceFacet*
  - *LiquidityFacet*
  - *PoolFacet*
  - *SettlementFacet*
  - *TipFacet*
  - *ClaimDIVALinearVesting*
  - *DIVAOracleTellor*

References

SCSVS G11: Code clarity
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x111-G11-Code-Clarity.md
Solidity Docs
https://docs.soliditylang.org/en/v0.8.19/style-guide.html#naming-conventions

6.4. Use proper error for non-existing pool

Status 2023-06-02	IMPLEMENTED

The recommendation has been implemented as proposed. The *_poolExists* function has been added and used in specified and other functions.

The *_setFinalReferenceValue* function now verifies whether the pool exists:

```
(...)
    function _setFinalReferenceValue(
        bytes32 _poolId,
        uint256 _finalReferenceValue,
        bool _allowChallenge,
        LibDIVAStorage.PoolStorage storage _ps,
        LibDIVAStorage.GovernanceStorage storage _gs
    ) private {
        // Initialize Pool struct
        LibDIVAStorage.Pool storage _pool = _ps.pools[_poolId];

        // Check if pool exists
        if (!LibDIVA._poolExists(_pool)) revert NonExistentPool();
```

```
(...)
```

The *_addTip* function now verifies whether the pool exists:

```
(...)
    function _addTip(
        bytes32 _poolId,
        uint256 _amount,
        LibDIVAStorage.PoolStorage storage _ps,
        LibDIVAStorage.FeeClaimStorage storage _fs
    ) private {
        // Load pool
        LibDIVAStorage.Pool storage _pool = _ps.pools[_poolId];

        // Check if pool exists
        if (!LibDIVA._poolExists(_pool)) revert NonExistentPool();
(...)
```

The *_checkAddLiquidityAllowed* function has been removed.

Severity

INFO

Description

There are functions that do not check whether the pool exists. These do not pose the risk as they revert in the further steps, but the revert error is not correct (e.g., instead of *PoolNotExisting* it reverts with a *PoolNotExpired* error).

The list of functions that should revert with *PoolNotExisting* error:

- Function *_setFinalReferenceValue* - SettlementFacet.sol#L411,
- Function *_addTip* - TipFacet.sol#L52,
- Function *_checkAddLiquidityAllowed* - LibDIVA.sol#L688.

Recommendation
Add new error *PoolNotExisting* and revert with this error in above-mentioned functions.

References

SCSVS G11: Code clarity
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x111-G11-Code-Clarity.md

6.5. Add incentive for the default settlement

Status 2023-06-02	NOT IMPLEMENTED

The team has acknowledged this recommendation but has no plans to implement it. The team assumes that the parties involved will be incentivized to get tokens anyway.

Severity

INFO

Description

The *_setFinalReferenceValue* function (SettlementFacet.sol#L411) is used to set the final value and settle the contingent pool. In case the data provider and fallback data provider do not report any values, anyone can settle the pool using the inflection value.

However, the settlement fee is then transferred to the treasury, instead of the *msg.sender*.

Recommendation
Transfer the settlement fee to the factual person who settles the pool to make it fair for position token holders and create an incentive to settle the pool.

References

SCSVS G4: Business logic
https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x100-General/0x104-G4-Business-Logic.md

6.6. Optimize gas consumption by removing redundant checks

Status 2023-06-02	IMPLEMENTED
The recommendation has been implemented as proposed. Redundant checks have been removed and comments on already met assumptions have been added in their place.	

Severity

INFO

Description

When designing projects based on smart contracts, it is particularly important to pay attention to the gas used. Consumption should be optimized as much as possible. Therefore,

there is no need for keeping redundant checks that are always satisfied in the contract. This has a direct impact on costs for users and, in exceptional cases, on security.

A helpful tool for estimating gas consumption and tracking optimization quality can be gas reporters:

- https://book.getfoundry.sh/forge/gas-reports
- https://www.npmjs.com/package/hardhat-gas-reporter

Recommendation

• Remove unnecessary checks:
  ◦ The check `if (_pool.statusFinalReferenceValue == LibDIVAStorage.Status.Confirmed)` is not needed as the pool's status is always Confirmed at this point (SettlementFacet.sol#L315).
  ◦ The check `block.timestamp > submissionEndTime` is not needed as in the *elseif* block it is always satisfied (SettlementFacet.sol#L475).
  ◦ The check `else if (_finalReferenceValue > _inflection) {` is not needed as in the *else* block it is always satisfied (LibDIVA.sol#L421).

References

Solidity gas optimization tips by devanshbatham:
https://github.com/devanshbatham/Solidity-Gas-Optimization-Tips
Solidity gas optimization tips by Mudit Gupta:
https://mudit.blog/solidity-gas-optimization-tips/
Under-Optimized Smart Contracts Devour Your Money research paper:
https://arxiv.org/pdf/1703.03994.pdf

6.7. Avoid zero value transfers initiated by the protocol

Status 2023-06-02	NOT IMPLEMENTED
The recommendation has not been implemented. The team believes that zero value transfers should be excluded on the frontend side rather than within the contract itself. *"Introducing the proposed check would result in additional gas costs. In particular, as we anticipate that data providers will utilize the batchClaimFee function, passing a collateral token with an amount of 0 by accident would cause the entire transaction to revert, leading to significant costs for the data provider."*	

Severity

INFO

Description

Function *_claimFee* (ClaimFacet.sol#L84) called indirectly via *claimFee* function (ClaimFacet.sol#L13) allows anyone to make a zero-value transfer of collateral token initiated by the pool. It might be used by scammers to legitimate scams.

We are aware of the fact that Etherscan lately decided to hide zero-value transfers by default, but this case is different from the common case of zero-transfers using *transferFrom* function. It makes a direct transfer from the pool using the *transfer* function.

Recommendation
Add the requirement for the claimed fee to be greater than 0.

References

SCSVS G1: Architecture, design and threat modeling
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x101-G1-Architecture-Design-Threat-Modeling.md

6.8. Detect duplicates in claimers' addresses

Status 2023-06-02	IMPLEMENTED
The recommendation has been implemented. The team updated the TS script that generates the Merkle Root for vesting and detects duplicates.	

Severity

INFO

Description

The *ClaimDIVALinearVesting* contract is used to claim the *DIVA* tokens. Its *claimTokens* function (ClaimDIVALinearVesting.sol#L45) allows the whitelisted addresses (verified using the Merkle Tree structure) to transfer out tokens. It is important to note that there must not be duplicates of addresses in the tree's leaves because the same addresses would not be able to withdraw tokens.

Recommendation

Make sure that there are no duplicate addresses in the list of accepted recipients.

References

SCSVS G2: Policies and procedures
https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x100-General/0x102-G2-Policies-procedures.md

6.9. Consider adding white hat hacks policy

Status 2023-06-02	NOT IMPLEMENTED
The recommendation has not been implemented. The team plans to implement such a policy after the mainnet launch.	

Severity

INFO

Description

There is a debate in the BlockSec community about how to approach hacks made by researchers with good intentions. In our opinion, the decision on this type of action is always up to the project team and the key aspect is to think about this issue before the situation arises.

The options you can choose from are:

1. Completely prohibit this kind of activity, regardless of intentions.
2. Use a hybrid approach that allows for this type of activity, but under strict rules.
3. Fully allow researchers to act freely if they have good intentions and the knowledge necessary to save funds.

Based on our experience, option 2 is the best especially if the project uses the services of the bug bounty platform. This allows you to delegate triaging and be informed only about confirmed cases.

Recommendation
• Include a policy covering the event of a white hat hack in the project security section and clearly define the rules that the researcher should follow. An example policy/statement might look like the following:

In case of detecting a vulnerability in production environment that threatens the security of the DIVA protocol, please follow the rules below:

1. Please report it through bug bounty platform with whom we work [LINK] and @tag [PERSON1] and [PERSON2] on the [BUG BOUNTY PLATFORM] #general channel that you urgently need contact.
2. If you do not receive a response within [MINUTES] minutes, send an e-mail with a description of the problem to [ADDRESS] encrypted with the following key [KEY].

We consent to the white hat hack ONLY after all the following conditions are met:

- You have the knowledge necessary to carry out a controlled attack (at least 3 years of professional experience in auditing smart contracts) and are fully aware of the threats associated with MEV bots.
- You successfully launched an attack on your local fork.
- You have performed steps 1-2 above and have not received ANY response for [MINUTES] minutes.

In the event of non-compliance with our guidelines, any attempt to hack the protocol will be prosecuted.

References

SCSVS G2: Policies and procedures
https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x100-General/0x102-G2-Policies-procedures.md

6.10. Remove *payable* mutability from *withdraw* function

Status 2023-06-02	NOT IMPLEMENTED

The recommendation has not been implemented.

The team decided to leave the code unchanged as funds can be withdrawn by the owner using the *withdrawDirectDeposit* function.

Severity

INFO

Description

The *withdraw* and *withdrawDirectDeposit* in *DIVADevelopmentFund* are marked as *payable* even though it is used for withdrawals and not deposits (*DIVADevelopmentFund.sol#L76*).

Although this is a kind of gas optimization and it is possible to extract funds accidentally deposited by this function we recommend to declare as *payable* only functions that receive ether.

```
74      function withdraw(address _token, uint256[] calldata _indices)
75          external
76          payable
77          override
78          nonReentrant
79          onlyDIVAOwner
80      {
```

```
110 function withdrawDirectDeposit(address _token)
111         external
112         payable
113         override
114         nonReentrant
115         onlyDIVAOwner
```

Recommendation
Remove *payable* from *withdraw* and *withdrawDirectDeposit* functions.

References

SCSVS G11: Code clarity
https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x100-General/0x111-G11-Code-Clarity.md

6.11. Consider adding popups for front-end application to warn users

Status 2023-06-02	NOT IMPLEMENTED
The recommendation has not been implemented.	

Severity

INFO

Description

The protocol gives users a lot of freedom, but with that comes a lot of responsibility. To increase awareness and make sure that they consciously take risks, we propose to highlight actions that potentially, despite the best efforts and proper security measures, may expose the users to loss.

In particular, it concerns operations such as the selection of token for contingent pool and data provider, which settles the result.

Recommendation
• When users decide to use a token other than from the predefined list (considered as verified), they should be notified and confirm their awareness of the risks associated with the use of untrusted tokens. • When users decide to use a data provider other than from the predefined list (considered as verified) or the pool that uses such data provider, they should be notified and confirm their awareness of the risks associated with the use of untrusted oracles. • When users decide to use a NFT token for permissioned pools other than from the predefined list (considered as verified), they should be notified and confirm their awareness of the risks associated with the use of untrusted tokens.

References

SCSVS G1: Architecture, design and threat modeling
https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x100-General/0x101-G1-Architecture-Design-Threat-Modeling.md

6.12. Consider extending the effect of the *pauseReturnCollateral* function

Status 2023-06-02	NOT IMPLEMENTED
The recommendation has not been implemented. The team decided to leave the code unchanged to prevent the owner from being pressured by a central authority to halt the entire protocol.	

Severity

INFO

Description

The purpose of the *pauseReturnCollateral* (GovernanceFacet.sol#L210) functionality is to minimize the potential harm in the event of a hack. According to the design, it does not block all operations that handle tokens, it focuses on returns of collateral.

However, it is a good practice to also prohibit further deposits to minimize the attack surface area when it is forbidden to withdraw tokens.

Recommendation
Extend the functionality of *pauseReturnCollateral* inside *_checkAddLiquidityAllowed* function to cover also further deposits by *addLiquidity*, *batchAddLiquidity*, *fillOfferCreateContingentPool* and *batchFillOfferCreateContingentPool*.

References

SCSVS G1: Architecture, design and threat modeling
https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x100-General/0x101-G1-Architecture-Design-Threat-Modeling.md

6.13. Protect withdrawing all tokens before setting up trigger

Status 2023-06-02	NOT IMPLEMENTED
The recommendation has not been implemented. The team decided to leave the code unchanged: *"Not addressed as this may be useful in case something goes wrong at initialization."*	

Severity

INFO

Description

The function *withdrawUnclaimedDivaTokens* (ClaimDIVALinearVesting.sol#L128) can be called by the owner after the transfer of all vested tokens and before the vesting is triggered.

In such a case, the following check is bypassed as the value of the *claimPeriodStarts* variable is *0*.

```
128 function withdrawUnclaimedDivaTokens() external onlyOwner {
129         require(
130             block.timestamp >= claimPeriodStarts.add(YEAR.mul(3)),
131     "ClaimDIVA: cant claim diva tokens until three years from claim start 132time"
132         );
133         divaToken.transfer(msg.sender, divaToken.balanceOf(address(this)));
134     }
```

Note: This has been reported as a recommendation because the dev team treats the trigger moment as the start of funds protection. That is because before that moment, the owner controls all tokens anyway.

Recommendation
Add additional requirement to the *withdrawUnclaimedDivaTokens* function that will allow calling this function only after the *trigger* variable has been set.

References

SCSVS G5: Access control
https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x100-General/0x105-G5-Access-Control.md

6.14. Use the same version of Solidity in all smart contracts (latest stable)

Status 2023-06-02	IMPLEMENTED
The recommendation has been implemented as proposed. Currently, all smart contracts use the same version, 0.8.19.	

Severity

INFO

Description

Not all contracts use version *0.8.19*. Some of them use version *0.8.9*.

Recommendation
Unify, and use version *0.8.19* for the following smart contracts: • ClaimDIVALinearVesting • DIVAOracleTellor • UsingTellor

References

SCSVS G11: Code clarity
https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x100-General/0x111-G11-Code-Clarity.md

7. Impact on risk classification

Risk classification is based on the one developed by OWASP, however it has been adapted to the immutable and transparent code nature of smart contracts. The Web3 ecosystem forgives much less mistakes than in the case of traditional applications, the servers of which can be covered by many layers of security.

Therefore, the classification is more strict and indicates higher priorities for paying attention to security.

Overall risk severity				
Impact on risk	HIGH	CRITICAL	MAJOR	MEDIUM
	MEDIUM	MEDIUM	MEDIUM	MINOR
	LOW	MINOR	MINOR	INFO
		LOW	MEDIUM	HIGH
		Exploitation conditions		

OWASP Risk Rating methodology:
https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

8. Long-term best practices

8.1. Use automated tools to scan your code regularly

It's a good idea to incorporate automated tools (e.g. slither) into the code writing process. This will allow basic security issues to be detected and addressed at a very early stage.

8.2. Perform threat modeling

Before implementing or introducing changes to smart contracts, perform threat modeling and think with your team about what can go wrong. Set potential targets of the attacker and possible ways to achieve them, keep it in mind during implementation to prevent bad design decisions.

Read our guide on how to implement it in your team:
https://composable-security.com/blog/threat-modeling-for-smart-contracts-best-step-by-step-guide/

8.3. Use Smart Contract Security Verification Standard

Use proven standards to maintain a high level of security for your contracts. Treat individual categories as checklists to verify the security of individual components. Expand your unit tests with selected checks from the list to be sure when introducing changes that they did not affect the security of the project.

Smart Contract Security Verification Standard:
https://github.com/ComposableSecurity/SCSVS

8.4. Discuss audit reports and learn from them

The best guarantee of security is the continuous development of team knowledge. To use the audit as effectively as possible, make sure that everyone in the team understands the mistakes made. Consider whether the detected vulnerabilities may exist in other places, audits always have a limited time and the developers know the code best.

8.5. Monitor your and similar contracts

Use the tools available on the market to monitor key contracts (e.g. the ones where user's tokens are kept). If you have used code from another project, monitor their contracts as well and introduce procedures to capture information about detected vulnerabilities in their code.

8.6. Take care of the action policies

Prepare an action process for important eventualities. Make sure it is clear who is responsible for what, and those responsible know what to do. Cover at least the following scenarios: detection of vulnerabilities in the production environment, project updates/expansion, hack.

SCSVS G2: Policies and procedures is a good source to start with https://github.com/ComposableSecurity/SCSVS/blob/master/2.0/0x100-General/0x102-G2-Policies-procedures.md

9. Contact

Damian Rusinek
Smart Contract Security Auditor
@drdr_zz
damian.rusinek@composable-security.com

Paweł Kuryłowicz
Smart Contract Security Auditor
@wh01s7
pawel.kurylowicz@composable-security.com