hexens × Defexa

Apr.23

SMART CONTRACT AUDIT REPORT FOR DEFEXA

CONTENTS

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

AUDIT LED BY

VAHE KARAPETYAN

Co-founder / CTO | Hexens

Audit Starting Date
24.04.2023

Audit Completion Date
01.05.2023

---

hexens × Defexa

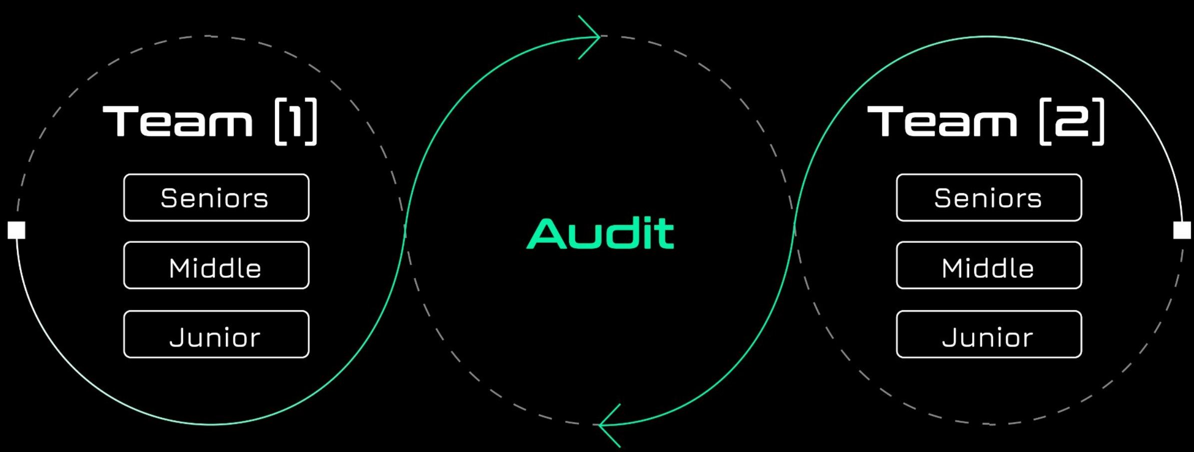
METHODOLOGY

COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

IMPACT	PROBABILITY			
	Rare	Unlikely	Likely	Very Likely
Low / Info	Low / Info	Low / Info	Medium	Medium
Medium	Low / Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of vulnerabilities:

CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of the project. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered Defexa's order protocol.

Our security assessment was a full review of Defexa's protocol and its smart contracts. We have thoroughly reviewed each contract individually and the system as a whole.

During the security assessment process, we uncovered 1 critical severity vulnerability in the DefexaExchange. It would allow an attacker to stop the protocol.

We have also identified 4 high severity vulnerabilities, various minor vulnerabilities, and code optimizations.

Finally, all of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality has increased after the completion of our audit.

SCOPE

The analyzed resources were sent in an archive with the following SHA256 hash:
397c94eb91f10c7c5551a2742fd6969dd98c8301664a527e8345711d3451e8ab

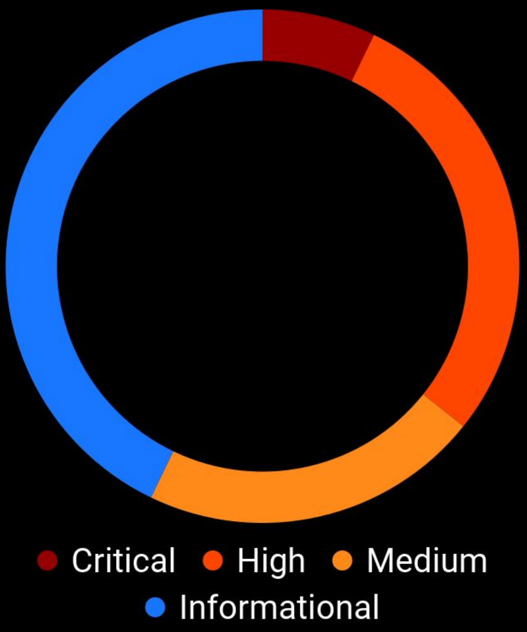
The issues described in the report were fixed in the following version (SHA256 hash):
84675d0cc3ba467e17cc6e56e193e94055fa9e25f261f63b4887822d70e4cc0f

SUMMARY

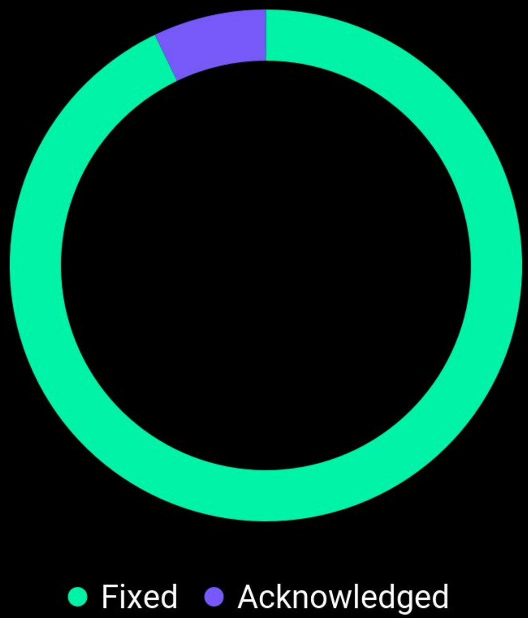
SEVERITY	NUMBER OF FINDINGS
CRITICAL	1
HIGH	4
MEDIUM	3
LOW	0
INFORMATIONAL	6

TOTAL: 14

SEVERITY



STATUS



WEAKNESSES

This section contains the list of discovered weaknesses.

DEX-3. ANYONE CAN STOP THE PROTOCOL

SEVERITY: Critical

PATH: DefexaExchange.sol, Living.sol

REMEDIATION: use OpenZeppelin's Pausable Library

STATUS: fixed

DESCRIPTION:

The **DefexaExchange.sol** contract imports **WardedLivingUpgradeable.sol**, which, in turn, imports **Living.sol**. **Living.sol** has an external **stop()** function, a public **run()** function, and a **live** modifier that requires **alive** to not equal 0. In the **DefexaExchange.sol** contract, the **alive** variable is set to 1 in the **initialize()** function, and the **live** modifier is used for the **createOrder** and **cancelOrder** functions. However, since anyone can call the functions **run()** and **stop()**, it means anyone can set **alive** equal to 0, causing the protocol to stop, or in case the protocol should be stopped, anyone can run it.

```
abstract contract Living {
  uint256 alive;

  modifier live {
      require(alive != 0, "Living/not-live");
      _;
  }

  function stop() external {
      alive = 0;
  }

  function run() public {
      alive = 1;
  }
}
```

DEX-13. BAD ACTORS CAN DRAIN MONEY FROM THE CONTRACT OR MANIPULATE IT

SEVERITY: High

PATH: Arbitrage.sol

REMEDIATION: implement access control, to prevent unauthorized actions and ensure the safety of the contract and its users. Consider using, for example, OpenZeppelin's Ownable.sol, or already implemented Warded.sol

STATUS: fixed

DESCRIPTION:

The **Arbitrage** smart contract is designed to facilitate the execution of multiple external function calls via its **multiCall()** function. The **multiCall()** and **approve()** functions in this contract are both marked as external and have no access modifiers. This presents a potential vulnerability, as a malicious actor could potentially manipulate the contract through these functions.

Specifically, the **approve()** function lacks security checks, which could enable a bad actor to exploit the contract by giving themselves or others allowance, and potentially manipulate the **multiCall()** function to draw funds from the contract.

```
contract Arbitrage {

  receive() external payable {}

  function multiCall(
     address[] calldata targets,
     bytes[] calldata data
  ) external payable {
     require(targets.length == data.length, "target length != data length");

     for (uint i; i < targets.length; i++) {
        (bool success,) = targets[i].call(data[i]);
        require(success, "call failed");
     }
  }
  function approve(address token, address spender, uint256 amount) external {
     IERC20(token).approve(spender, amount);
  }
}
```

DEX-2. BAD ACTOR CAN FRONTRUN ORDERS MATCHING

SEVERITY: High

PATH: DefexaExchange.sol

REMEDIATION: use a secure random number generator or a nonce-based approach to generate unique _orderId

STATUS: fixed

DESCRIPTION:

In the function createOrder() the current method of generating a new order ID is by hashing the sender **address**, **amount**, and current **block.timestamp**, this implementation is not secure. A malicious actor can create an order for swapping TokenA to TokenB. Then if it matches in the same block, they can frontrun and create another order with changed **price**, **_isQuote**, and **_orderType** fields but with the same **_orderId**, potentially causing confusion or manipulation.

```
function createOrder(
    address _tokenA,
    address _tokenB,
    uint256 _amount,
    uint256 _price,
    bool _isQuote,
    uint8 _orderType
  ) external payable override live returns (uint256) {
    if (_tokenA == _tokenB) {
      revert TokensMismatch();
    }
    if (_orderType != ORDER_TYPE_GTC) {
      revert OrderTypeNotSupported(_orderType);
    }

    uint256 holdAmount = _amount;
    if (_isQuote) {
      holdAmount = (_amount * _price) / 1e18;
    }

    if (_tokenA == address(0) && msg.value != holdAmount) {
      revert InvalidOrder();
    }

    uint256 newId = uint256(
      keccak256(abi.encode(msg.sender, block.timestamp, _amount))
    );
    orders[newId] = Order({
      id: newId,
      createdAt: block.timestamp,
      user: msg.sender,
      tokenA: _tokenA,
      tokenB: _tokenB,
      amount: _amount,
      initialAmount: holdAmount,
      spentAmount: 0,
      price: _price,
      isQuote: _isQuote,
      orderType: _orderType,
      status: ORDER_STATUS_NEW
    });
```

```
    if (_tokenA != address(0)) {
      if (
        !IERC20(_tokenA).transferFrom(
          msg.sender,
          address(this),
          holdAmount
        )
      ) {
        revert TransferFailed();
      }
    }

    emit NewOrder(
      msg.sender,
      newId,
      _amount,
      _price,
      _tokenA,
      _tokenB,
      block.timestamp,
      _isQuote,
      _orderType
    );

    return newId;
  }
```

DEX-14. ANYONE CAN UPGRADE THE PROTOCOL

SEVERITY: High

PATH: DefexaVault.sol

REMEDIATION: add onlyOwner modifier

STATUS: fixed

DESCRIPTION:

The **_authorizeUpgrade** function is missing an owner check, which means that anyone can perform a proxy upgrade and potentially steal funds from the contract.

```
function _authorizeUpgrade(address newImplementation) internal override {}
```

DEX-7. INCORRECT IF STATEMENT

SEVERITY: High

PATH: DefexaExchange.sol

REMEDIATION: remove if (taker.isQuote)

STATUS: fixed

DESCRIPTION:

In the **_fill** function, there is a check to ensure that **taker.price** is not greater than **maker.price**, but this check only applies if **taker.isQuote** is set to **true**. However, a user can create an order with the **taker.isQuote** parameter set to **false** thus, the if-statement that is designed to protect the use will be bypassed. As a result, such an order can be matched with other orders that have a higher price.

```
function _fill(
    Order storage maker,
    Order storage taker
) internal {
    if (taker.isQuote) {
        if (1e36 / taker.price > maker.price) {
            revert PriceMismatch(maker.price, taker.price);
        }
    }

    (uint256 makerAmount, uint256 makerQuote, uint256 takerAmount, uint256 takerQuote) =
        _getAmountForPrice(maker, taker);
    if (makerQuote > takerAmount) {
        makerQuote = takerAmount;
    }
    if (takerQuote > makerAmount) {
        takerQuote = makerAmount;
    }
    uint256 takerToMaker = makerQuote;
    uint256 makerToTaker = takerQuote;

    if (maker.isQuote) {
        maker.amount -= takerToMaker;
    } else {
        maker.amount -= makerToTaker;
    }
    maker.spentAmount += makerToTaker;
    if (taker.isQuote) {
        taker.amount -= makerToTaker;
    } else {
        taker.amount -= takerToMaker;
    }
    taker.spentAmount += takerToMaker;

    _setOrderStatus(maker);
    _setOrderStatus(taker);

    uint256 fee = _takeFee(taker.id, makerToTaker, taker.user, taker.tokenB);
    _returnLeftover(taker);
    _returnLeftover(maker);

    _send(taker.user, maker.tokenA, makerToTaker - fee);
    _send(maker.user, taker.tokenA, takerToMaker);

    _emitOrderFilled(maker, makerToTaker, takerToMaker, 0);
    _emitOrderFilled(taker, takerToMaker, makerToTaker, fee);
}
```

DEX-10. INCOMPLETE IMPLEMENTATION OF LOGIC

SEVERITY: Medium

PATH: DefexaExchange.sol

REMEDIATION: add the implementation of a given logic

STATUS: acknowledged

DESCRIPTION:

The protocol specifies that maker orders should be sorted from best to worst price, but there are no checks or implementations in place to ensure this. Thus the authorized caller can, in fact, match orders with any prices in a centralized manner.

```
// @dev
// match taker order with makers
// maker orders should be ordered from the best price to worse
function matchOrders(
    uint256[] memory _makers,
    uint256 _takerId
) public override auth {
    if (orders[_takerId].createdAt == 0) {
        revert OrderNotFound(_takerId);
    }
    if (
        orders[_takerId].status != ORDER_STATUS_NEW &&
        orders[_takerId].status != ORDER_STATUS_PARTIALLY_FILLED
    ) {
        revert OrderStatusInvalid(orders[_takerId].status);
    }

    for (uint256 i = 0; i < _makers.length; i++) {
        Order storage taker = orders[_takerId];
        Order storage maker = orders[_makers[i]];
        if (maker.createdAt == 0) {
            revert OrderNotFound(_makers[i]);
        }
        if (
            orders[_makers[i]].status != ORDER_STATUS_NEW &&
            orders[_makers[i]].status != ORDER_STATUS_PARTIALLY_FILLED
        ) {
            revert OrderStatusInvalid(orders[_takerId].status);
        }
        if ((maker.tokenA != taker.tokenB) ||
            (maker.tokenB != taker.tokenA)) {
            revert TokensMismatch();
        }
        // console.log("[M1] TakerAmount: ", taker.amount);
        if (taker.amount == 0) {
            break;
        }
        _fill(maker, taker);
    }
}
```

DEX-9. MISSING LIMIT ON FEE

SEVERITY: Medium

PATH: DefexaExchange.sol

REMEDIATION: add a maximal fee size, e.g. 10% and consider checking that the new fee is less than that maximum fee

STATUS: fixed

DESCRIPTION:

There is no check to ensure that the set fee is not greater than 100%. E.g. an authorized person could set it to 150% and thereby drain money from the users.

```
function setTakerFee(uint256 _newFee) external auth {
    takerFee = _newFee;
    emit TakerFeeUpdated(_newFee, block.timestamp);
```

DEX-6. CREATEORDER FUNCTION PARAMETERS LACK ZERO VALUE CHECK

SEVERITY: Medium

PATH: DefexaExchange.sol

REMEDIATION: add checks for zero values for _amount and _price input parameters

STATUS: fixed

DESCRIPTION:

The function **createOrder()** accepts **_amount** and **_price** as input parameters, but it lacks a check for zero values. This vulnerability could be exploited by a malicious actor to create a large number of orders, causing issues for other users in finding and matching corresponding orders or causing issues in the front end, spamming with orders. Additionally, even without a check on the **_amount** parameter, a bad actor could still call the **matchOrders()** function.

```
function createOrder(
    address _tokenA,
    address _tokenB,
    uint256 _amount,
    uint256 _price,
    bool _isQuote,
    uint8 _orderType
) external live payable override returns (uint256) {
    if (_tokenA == _tokenB) {
        revert TokensMismatch();
    }
    if (_orderType != ORDER_TYPE_GTC) {
        revert OrderTypeNotSupported(_orderType);
    }

    uint256 holdAmount = _amount;
    if (_isQuote) {
        holdAmount = _amount * _price / 1e18;
    }

    if (_tokenA == address(0) && msg.value != holdAmount) {
        revert InvalidOrder();
    }

    uint256 newId = uint256(keccak256(abi.encode(msg.sender, block.timestamp, _amount)));
    orders[newId] = Order({
        id: newId,
        createdAt: block.timestamp,
        user: msg.sender,
        tokenA: _tokenA,
        tokenB: _tokenB,
        amount: _amount,
        initialAmount: holdAmount,
        spentAmount: 0,
        price: _price,
        isQuote: _isQuote,
        orderType: _orderType,
        status: ORDER_STATUS_NEW
    });
```

```
        if (_tokenA != address(0)) {
            if (!IERC20(_tokenA).transferFrom(msg.sender, address(this), holdAmount)) {
                revert TransferFailed();
            }
        }

        emit NewOrder(msg.sender, newId, _amount, _price, _tokenA, _tokenB, block.timestamp, _isQuote, _orderType);

        return newId;
    }
```

DEX-8. BETTER WORKING COMPARISON LOGIC

SEVERITY: Informational

PATH: DefexaExchange.sol

REMEDIATION: change 1e36 / taker.price > maker.price to 1e36 > maker.price * taker.price

STATUS: fixed

DESCRIPTION:

In the **_fill** function, the current price check is **1e36 / taker.price > maker.price** (L126). However, it is recommended to change it to **1e36 > maker.price * taker.price** for better readability and potential rounding errors.

```
function _fill(
    Order storage maker,
    Order storage taker
) internal {
    if (taker.isQuote) {
        if (1e36 / taker.price > maker.price) {
            revert PriceMismatch(maker.price, taker.price);
        }
    }

    (uint256 makerAmount, uint256 makerQuote, uint256 takerAmount, uint256 takerQuote) =
        _getAmountForPrice(maker, taker);
    if (makerQuote > takerAmount) {
        makerQuote = takerAmount;
    }
    if (takerQuote > makerAmount) {
        takerQuote = makerAmount;
    }
    uint256 takerToMaker = makerQuote;
    uint256 makerToTaker = takerQuote;

    if (maker.isQuote) {
        maker.amount -= takerToMaker;
    } else {
        maker.amount -= makerToTaker;
    }
    maker.spentAmount += makerToTaker;
    if (taker.isQuote) {
        taker.amount -= makerToTaker;
    } else {
        taker.amount -= takerToMaker;
    }
    taker.spentAmount += takerToMaker;

    _setOrderStatus(maker);
    _setOrderStatus(taker);

    uint256 fee = _takeFee(taker.id, makerToTaker, taker.user, taker.tokenB);
    _returnLeftover(taker);
    _returnLeftover(maker);

    _send(taker.user, maker.tokenA, makerToTaker - fee);
    _send(maker.user, taker.tokenA, takerToMaker);

    _emitOrderFilled(maker, makerToTaker, takerToMaker, 0);
    _emitOrderFilled(taker, takerToMaker, makerToTaker, fee);
}
```

DEX-4. FUNCTION PARAMETER LACKS ZERO ADDRESS CHECK

SEVERITY: Informational

PATH: DefexaExchange.sol, DefexaVault.sol

REMEDIATION: add zero address checks to _feeCollector variable

STATUS: fixed

DESCRIPTION:

In functions **initialize() address _feeCollector** lacks a zero address check.

In function **setFeeCollector() address _feeCollector** lacks a zero address check.

```
function initialize(
    address _feeCollector,
    uint256 _takerFee
) public initializer {
    __Ownable_init();
    __WardedLiving_init();

    feeCollector = _feeCollector;
    takerFee = _takerFee;
}

[..]

function setFeeCollector(address _feeCollector) external auth {
    feeCollector = _feeCollector;
    emit FeeCollectorUpdated(feeCollector, block.timestamp);
}
```

DEX-5. REDUNDANT IF STATEMENTS

SEVERITY: Informational

PATH: DefexaExchange.sol

REMEDIATION: remove that if statements

STATUS: fixed

DESCRIPTION:

The functions **cancelOrder** and **_returnLeftover** include an **if** statement that checks if **leftover > 0**. However, in **cancelOrder** this function only works if the order status is either **ORDER_STATUS_NEW** or **ORDER_STATUS_PARTIALLY_FILLED**. If the order status is one of the mentioned, it already implies that **orders[_orderId].initialAmount - orders[_orderId].spentAmount**, which is **leftover** is greater than 0.

In **_returnLeftover leftover** also should be greater than 0.

In the case that **leftover** is equal to 0, **_send** function will be called with 0 amount, which also has no impact.

```
function cancelOrder(
    uint256 _orderId
) external live override {
    if (orders[_orderId].status != ORDER_STATUS_NEW &&
        orders[_orderId].status != ORDER_STATUS_PARTIALLY_FILLED) {
        revert OrderStatusInvalid(orders[_orderId].status);
    }
    if (orders[_orderId].user != msg.sender) {
        revert Forbidden();
    }

    orders[_orderId].status = ORDER_STATUS_CANCELLED;

    uint256 leftover = orders[_orderId].initialAmount - orders[_orderId].spentAmount;
    if (leftover > 0) {
        _send(msg.sender, orders[_orderId].tokenA, leftover);
    }

    emit OrderCanceled(msg.sender, _orderId, block.timestamp);
}
```

DEX-11. IMPROVE CODE READABILITY

SEVERITY: Informational

PATH: DefexaExchange.sol

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The **matchOrders** function currently includes two checks for **OrderNotFound** and **OrderStatusInvalid** for both **taker** and **makers**. To improve code readability, these checks can be consolidated into a separate function f.e called **isValidOrder()**, or some modifier.

```
[..]
    if (orders[_takerId].createdAt == 0) {
        revert OrderNotFound(_takerId);
    }
    if (
        orders[_takerId].status != ORDER_STATUS_NEW &&
        orders[_takerId].status != ORDER_STATUS_PARTIALLY_FILLED
    ) {
        revert OrderStatusInvalid(orders[_takerId].status);
    }

    for (uint256 i = 0; i < _makers.length; i++) {
        Order storage taker = orders[_takerId];
        Order storage maker = orders[_makers[i]];
        if (maker.createdAt == 0) {
            revert OrderNotFound(_makers[i]);
        }
        if (
            orders[_makers[i]].status != ORDER_STATUS_NEW &&
            orders[_makers[i]].status != ORDER_STATUS_PARTIALLY_FILLED
        ) {
            revert OrderStatusInvalid(orders[_takerId].status);
        }
[..]
```

DEX-1. FUNCTION'S STATE MUTABILITY

SEVERITY: Informational

PATH: DefexaExchange.sol

REMEDIATION: change view to pure for clean code purposes

STATUS: fixed

DESCRIPTION:

The state mutability of a function can be changed from **view** to **pure**.

```
function _getAmountForPrice(
    Order memory maker,
    Order memory taker
) internal view returns(
    uint256 makerAmount, uint256 makerQuote, uint256 takerAmount, uint256 takerQuote) {
    takerQuote = taker.amount * maker.price / 1e18;
    makerQuote = maker.amount * 1e18 / maker.price;
    makerAmount = maker.amount;
    takerAmount = taker.amount;
    if (maker.isQuote) {
        makerQuote = maker.amount;
        makerAmount = maker.amount * maker.price / 1e18;
    }
    if (taker.isQuote) {
        takerQuote = taker.amount;
        takerAmount = taker.amount * 1e18 / maker.price;
    }
}
```

DEX-12. REDUNDANT RETURN

SEVERITY: Informational

PATH: DefexaExchange.sol

REMEDIATION: remove return leftover;

STATUS: fixed

DESCRIPTION:

The **return leftover;** statement in line 265 is redundant since the function declaration already specifies that the function returns a uint256 with the name leftover: **returns(uint256 leftover)**.

```
function _returnLeftover(Order memory taker) internal returns(uint256 leftover) {
    if (taker.isQuote && taker.amount == 0) {
        leftover = taker.initialAmount - taker.spentAmount;
        if (leftover > 0) {
            _send(taker.user, taker.tokenA, leftover);

            emit LeftoverReturned(taker.user, taker.id, taker.tokenA, leftover, block.timestamp);
        }
    }

    return leftover;
}
```

hexens