hexens x UFARM DIGITAL

Security Review Report for UFarm

May 2025

Table of Contents

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This report covers the audit of UFarm protocol, an on-chain digital asset management solution. The audit included updates to the core protocol code in comparison to the previous version.

Our security assessment was a full review of the changes in scope, spanning a total of 2 weeks.

During our audit, we identified 1 critical severity vulnerability, which could have permanently blocked the protocol, and 3 high severity vulnerabilities.

We also identified several minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed by the development team and consequently validated by us.

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

3. Security Review Details

▪ Review Led by

Kasper Zwijsen, Head of Audits

▪ Scope

The analyzed resources are located on:

🔗 https://gitlab.com/mobileup/ufarm-digital/ufarm-evm-contracts/-/tree/

📌 Commit: `aa69668de34c7bcd32cb271d082a4398d127b145`

The issues described in this report were fixed in the following commits:

🔗 https://github.com/UFarmDigital/UFarm-EVM-Contracts

📌 Commit: `2fc58b7b810b82a2385ea8e275665311e3b8364f`

▪ Changelog

26 May 2025	Audit start
10 June 2025	Initial report
15 June 2025	Revision received
30 June 2025	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract 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 smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

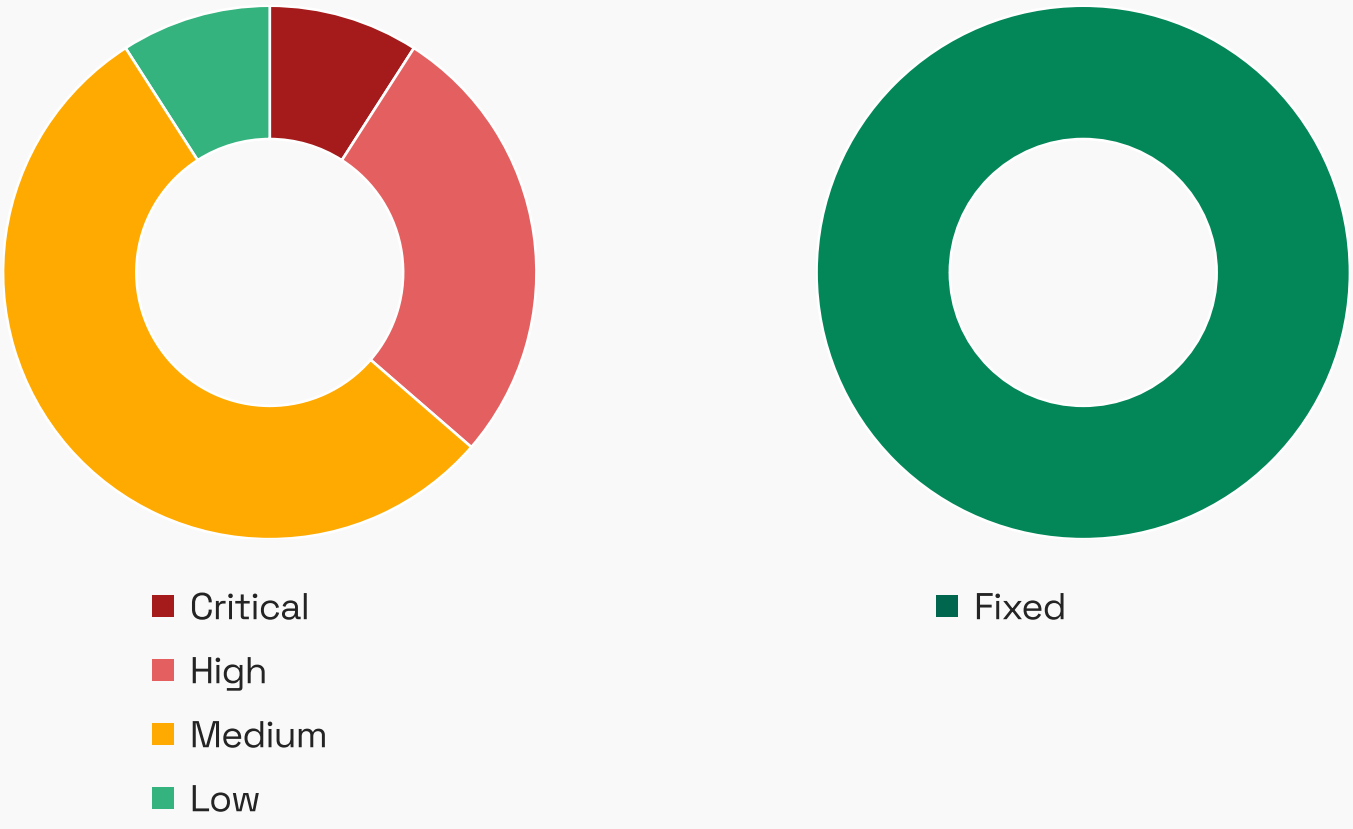
- **Issue Symbolic Codes**

Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
Critical	1
High	3
Medium	5
Low	1
Informational	0
Total:	**10**



6. Weaknesses

This section contains the list of discovered weaknesses.

UFARM1-2 | Quex callback can be permanently DoSed by congesting queues

Fixed

Severity:	Probability:	Impact:
Critical	Likely	Critical

Path:

main/contracts/pool/UFarmPool.sol:quexCallback#L881-L1007

Description:

The UFarmPool contract uses the Quex Oracle to query off-chain information, namely the total value of the pool (**_totalCost**).

Any deposit or withdrawal action will add this action to the corresponding queue (**depositQueue** and **withdrawQueue**) and fire a Quex request. The Quex core contract will call into **quexCallback** to resolve the request with the queried data.

The issue here however, is that **quexCallback** contains unbounded loops in the handling of the queues. More specifically, it will loop over the entire **depositQueue** and **withdrawQueue**, and execute each action in there:

```
requestsLength = depositQueue.length;
while (requestsLength > 0) {
    [..]
}
```

An attacker can arbitrarily create a large amount of deposit requests without the need for assets (the tokens are only transferred in the callback). As a result, the **quexCallback** will always revert with out-of-gas. The Quex flow currently has a constant gas limit of 1m gas, which can be easily filled.

Moreover, in the current implementation the deposit queue and withdraw queue can only be reduced/popped inside of the callback. So if it (forcibly) grows outside of maximum gas capabilities, then the pool would have no way of recovering from the DoS attack.

```
function quexCallback(uint256 receivedRequestId, DataItem memory response)
external {
    [..]
    {
        uint256 sharesToMint;
        uint256 amountToInvest;
        uint256 totalDeposit;
        bytes32 depositRequestHash;
        QueueItem memory depositItem;

        requestsLength = depositQueue.length;
        while (requestsLength > 0) {
            // Validate each deposit request
            depositItem = depositQueue[requestsLength - 1];
            amountToInvest = depositItem.amount;
            investor = depositItem.investor;
            depositRequestHash = depositItem.requestHash;

            // Process the deposit
            try this.safeTransferToPool(investor, amountToInvest,
depositItem.bearerToken) {
                sharesToMint = _mintSharesByQuote(investor, amountToInvest,
_totalCost);

                // Adjust the total cost and total deposit
                _totalCost += amountToInvest;
                totalDeposit += amountToInvest;

                emit Deposit(investor, depositItem.bearerToken, amountToInvest,
sharesToMint);

                if (depositRequestHash != bytes32(0)) {
                    __usedDepositsRequests[depositRequestHash] = true;
                    emit DepositRequestExecuted(investor, depositRequestHash);
                }
            } catch {
                depositQueue.pop();
                requestsLength = depositQueue.length;
                continue;
            }

            depositQueue.pop();
            requestsLength = depositQueue.length;
```

```
            }

            highWaterMark += totalDeposit;
        }
        [..]
    }
```

Remediation:

The **quexCallback** function should better manage gas limitations by only handling a maximum number of deposit/withdraw requests.

UFARM1-1 | The approve top-up role can replay the same deposit/withdrawal request

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

contracts/main/contracts/pool/UFarmPool.sol#L404-L415
contracts/main/contracts/pool/UFarmPool.sol#L451-L461

Description:

The **UFarmPool::approveDeposits()** function allows the approve top-up role (**Permissions.Pool.ApprovePoolTopup & Permissions.Fund.ApprovePoolTopup**) to approve multiple deposit requests. It validates each request and pushes them to the **depositQueue**, then executes all requests in that queue using **sendQuexRequest()**.

```
for (uint256 i; i < requestsLength; ++i) {
    try this.validateDepositRequest(_depositRequests[i]) returns (
        address investor,
        uint256 amountToInvest,
        bytes32 depositRequestHash,
        address bearerToken
    ) {
        depositQueue.push(QueueItem(amountToInvest, depositRequestHash,
investor, bearerToken));
    } catch {
        continue;
    }
}
sendQuexRequest();
```

However, there is no check for duplicate requests in the loop, since the validation in the **validateDepositRequest()** function only verifies whether a request has been completed.

```
if (__usedDepositsRequests[depositRequestHash]) revert
UFarmErrors.ActionAlreadyDone();
```

When all deposit requests in the **depositQueue** are executed in the **quexCallback()** function, there is no validation to prevent a completed request from being executed again. As a result, a deposit request may be executed multiple times, pulling more funds from the user than expected. The deposit top-up role could exploit this behavior to misuse user funds if the user has approved more than necessary.

```
while (requestsLength > 0) {
    // Validate each deposit request
    depositItem = depositQueue[requestsLength - 1];
    amountToInvest = depositItem.amount;
    investor = depositItem.investor;
    depositRequestHash = depositItem.requestHash;

    // Process the deposit
    try this.safeTransferToPool(investor, amountToInvest,
depositItem.bearerToken) {
        sharesToMint = _mintSharesByQuote(investor, amountToInvest,
_totalCost);

        // Adjust the total cost and total deposit
        _totalCost += amountToInvest;
        totalDeposit += amountToInvest;

        emit Deposit(investor, depositItem.bearerToken, amountToInvest,
sharesToMint);

        if (depositRequestHash != bytes32(0)) {
            __usedDepositsRequests[depositRequestHash] = true;
            emit DepositRequestExecuted(investor, depositRequestHash);
        }
    } catch {
        depositQueue.pop();
        requestsLength = depositQueue.length;
        continue;
    }

    depositQueue.pop();
    requestsLength = depositQueue.length;
}
```

Similarly, the withdrawal top-up role can approve the same withdrawal request multiple times using **approveWithdrawals()**, potentially withdrawing more of the user's shares than expected.

Remediation:

There should be validation to prevent repeated requests in **approveDeposits()** and **approveWithdrawals()**, or a check for the completion status when executing requests in **quexCallback()**.

UFARM1-4 | Malicious User Could Deactivate Pool

Fixed

Severity:	Probability:	Impact:
High	Very likely	Medium

Path:

contracts/main/contracts/pool/UFarmPool.sol#L496-L540

Description:

In the **UFarmPool:withdraw** function, the contract records a withdrawal request by storing the current block timestamp:

```
if (pendingWithdrawalsRequests[withdrawalRequestHash] == 0) {
    // Set the withdrawal request timestamp
    pendingWithdrawalsRequests[withdrawalRequestHash] = block.timestamp;
    emit WithdrawRequestReceived(investor, withdrawalRequestHash,
block.timestamp);
} else {
```

Later, when the callback is executed, the contract removes the request hash in two scenarios:

1. User has insufficient shares:

```
if (sharesToBurn > availableToWithdraw) {
    delete pendingWithdrawalsRequests[withdrawalRequestHash];

    withdrawQueue.pop();
    requestsLength = withdrawQueue.length;
    continue;
}
```

2. User withdraws a non-zero amount:

```
if (investor != ufarmFund && amountToWithdraw != 0) {
    // Mark the request as used
    __usedWithdrawalsRequests[withdrawalRequestHash] = true;

    // Delete the request from the pending withdrawals
    delete pendingWithdrawalsRequests[withdrawalRequestHash];
}
```

The issue arises when **amountToWithdraw = 0**, their **withdrawalRequestHash** is not deleted. This creates a problem:

A malicious user can call the **withdraw** function again after the lockup period, using the same request hash, which forces the pool into a "deactivating" state.

```
if (config.withdrawalLockupPeriod > 0) {
    if (pendingWithdrawalsRequests[withdrawalRequestHash] == 0) {
        // Set the withdrawal request timestamp
        pendingWithdrawalsRequests[withdrawalRequestHash] = block.timestamp;
        emit WithdrawRequestReceived(investor, withdrawalRequestHash, block.timestamp);
    } else {
        // Check if the lockup period has passed
        uint256 unlockTime = pendingWithdrawalsRequests[withdrawalRequestHash] +
            config.withdrawalLockupPeriod;
        if (block.timestamp < unlockTime) {
            // Safe because of the check above
            revert LockupPeriodNotPassed(unlockTime);
        } else {
 @====>         _changeStatus(PoolStatus.Deactivating);
        }
    }
    return;
}
```

Remediation:

Consider deleting the request even when 0 shares are removed.

```
-- if (investor != ufarmFund && amountToWithdraw != 0) {
++ if (investor != ufarmFund) {
    // Mark the request as used
    __usedWithdrawalsRequests[withdrawalRequestHash] = true;

    // Delete the request from the pending withdrawals
    delete pendingWithdrawalsRequests[withdrawalRequestHash];
}
```

UFARM1-9 | Investor withdrawals may not function with a nonzero withdrawalLockupPeriod

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

contracts/main/contracts/pool/UFarmPool.sol#L517

Description:

UFarmPool::withdraw allows for a permissionless two-step withdrawal process for non-member investors if a nonzero **withdrawalLockupPeriod** is set in the pool's config. In this case, **_withdrawalRequests** parameters are first validated before a uniqueness check is performed on the request's hash. If the hash is unique, then the hash and current **block.timestamp** are saved in the **pendingWithdrawalsRequests** mapping in the first call to withdraw. On the second call, the request's unlock time is calculated. If the unlock time has passed, the pool's status is changed to deactivating.

By design, in either step of the withdrawal, the function will return before the withdrawal request is added to the withdrawal queue and **sendQuexRequest** is called. **ARCHITECTURE.md** states that after this, "the actual withdrawal will be queued on a subsequent call or by the fund manager once assets are ready".

However, this means that as long as **config.withdrawalLockupPeriod** is above zero, an investor's withdrawal calls will always return before the request contents are queued. As **withdrawQueue.push** on line 537 can never be reached, the investor's actual withdrawal request values are never recorded. This effectively prevents investors from withdrawing if **config.withdrawalLockupPeriod > 0**, requiring the pool's config to be manually changed.

```
//contracts/main/contracts/pool/UFarmPool.sol#L496-L540
...
    function withdraw(
        SignedWithdrawalRequest calldata _withdrawalRequest
    ) external override ufarmIsNotPaused nonReentrant {
        _checkStatusForFinancing(false);
        IPoolAdmin.PoolConfig memory config =
IPoolAdmin(poolAdmin).getConfig();

        if (msg.sender == ufarmFund) {
            withdrawQueue.push(
                QueueItem(
```

```
                    _withdrawalRequest.body.sharesToBurn,
                    keccak256(abi.encode(blockhash(block.number),
totalSupply())),
                    msg.sender,
                    valueToken
                )
            );
        } else {
            uint256 sharesToBurn;
            address investor;
            bytes32 withdrawalRequestHash;
            (investor, sharesToBurn, withdrawalRequestHash) =
validateWithdrawalRequest(_withdrawalRequest);

            if (config.withdrawalLockupPeriod > 0) {
                if (pendingWithdrawalsRequests[withdrawalRequestHash] == 0) {
                    // Set the withdrawal request timestamp
                    pendingWithdrawalsRequests[withdrawalRequestHash] =
block.timestamp;
                    emit WithdrawRequestReceived(investor,
withdrawalRequestHash, block.timestamp);
                } else {
                    // Check if the lockup period has passed
                    uint256 unlockTime =
pendingWithdrawalsRequests[withdrawalRequestHash] +
                        config.withdrawalLockupPeriod;
                    if (block.timestamp < unlockTime) {
                        // Safe because of the check above
                        revert LockupPeriodNotPassed(unlockTime);
                    } else {
                        _changeStatus(PoolStatus.Deactivating);
                    }
                }
                return;
            }

            withdrawQueue.push(QueueItem(sharesToBurn, withdrawalRequestHash,
investor, valueToken));
        }

        sendQuexRequest();
    }
...
```

Remediation:

As withdrawals are permitted regardless of pool status and the first investor to pass the unlock time changes the pool's overall status to deactivating, consider either of the following mitigations:

To preserve the 2-step nature of investor withdrawals after a pool status has been set to deactivating, consider queueing the withdrawal if the pool status is already deactivating on the second call with withdrawal, similar to the following:

```
//contracts/main/contracts/pool/UFarmPool.sol#L496-L540
...
    function withdraw(
        SignedWithdrawalRequest calldata _withdrawalRequest
    ) external override ufarmIsNotPaused nonReentrant {
        ...
            if (config.withdrawalLockupPeriod > 0) {
                ...
                    if (block.timestamp < unlockTime) {
                        // Safe because of the check above
                        revert LockupPeriodNotPassed(unlockTime);
                    } else {
++                      if (status == PoolStatus.Deactivating) {
++                          withdrawQueue.push(QueueItem(sharesToBurn,
withdrawalRequestHash, investor, valueToken));
++                          sendQuexRequest();
++                      }
++                      else {
++                          _changeStatus(PoolStatus.Deactivating);
                        }
                    }
                }
                return;
            }
            withdrawQueue.push(QueueItem(sharesToBurn, withdrawalRequestHash,
investor, valueToken));
        }

        sendQuexRequest();
    }
...
```

Alternatively, only execute the two-step check if the pool's status is not deactivating given a lockup period. Note that this will effectively nullify other investor unlock times once the pool is in deactivation mode.

```
//contracts/main/contracts/pool/UFarmPool.sol#L496-L540
...
    function withdraw(
        SignedWithdrawalRequest calldata _withdrawalRequest
    ) external override ufarmIsNotPaused nonReentrant {
        ...
--            if (config.withdrawalLockupPeriod > 0) {
++            if (config.withdrawalLockupPeriod > 0 && (status !=
PoolStatus.Deactivating)) {
              ...
              }
              return;
            }

            withdrawQueue.push(QueueItem(sharesToBurn, withdrawalRequestHash,
investor, valueToken));
        }

        sendQuexRequest();
    }
...
```

UFARM1-3 | Ques Callback Iterates Over Deposit and Withdraw Queues Backwards

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

contracts/main/contracts/pool/UFarmPool.sol#L826-L829

Description:

In **UFarmPool:quexCallback**, the queues for deposits and withdrawals are processed, however they are handled in reverse order, from the most recent to the oldest. This means the original order in which users made their requests is not respected.

```
requestsLength = depositQueue.length;
while (requestsLength > 0) {
    // Validate each deposit request
    depositItem = depositQueue[requestsLength - 1];
```

If the remediation for **UFARM1-2** such as introducing a rate limiter, is implemented, there would still be an unfair order of processing if requests continue to be handled in reverse. To ensure fairness, the system should process requests in the order they were received.

Remediation:

Handle the depositQueue and withdrawQueue in chronological order.

UFARM1-6 | Protocol fees and management fees have not been deducted from the profit in the calculation of the performance fee

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

contracts/main/contracts/pool/UFarmPool.sol#L748-L772

Description:

In the UFarmPool contract, **protocolFee** and **managementFee** are always charged whenever **_accrueFee** is called, regardless of whether the pool's **totalCost** is in profit or loss. These fees accumulate over time.

On the other hand, the **performanceFee** is only charged when the pool achieves a new profit, specifically when **totalCost > highWaterMark**. In this case, a percentage of the profit is taken as commission.

However, the current profit calculation does not deduct **protocolFee** and **managementFee**, resulting in a higher-than-actual profit being used to compute performance fees.

This happens because **protocolFee** and **managementFee** are continuously accrued over time to mint shares for UFarm's Core and UFarm's funds. As such, they can be considered as a cost or loss to the pool. Therefore, to accurately calculate profit, the **totalCost** of the pool should exclude these fees

```
{
    uint256 protocolCommission = IUFarmCore(_ufarmCore).protocolCommission();
    uint256 costInTime = (totalCost * accrualTime) / YEAR;

    (protocolFee, managementFee) = (
        (costInTime * protocolCommission) / ONE,
        (costInTime * managementCommission) / ONE
    );
}

if (totalCost > highWaterMark) {
    uint256 profit = totalCost - highWaterMark;

    performanceFee = (profit * PerformanceFeeLib.ONE_HUNDRED_PERCENT) /
highWaterMark; // APY ratio
```

```
    uint16 performanceCommission = performanceFee >
PerformanceFeeLib.MAX_COMMISSION_STEP
        ? PerformanceFeeLib.MAX_COMMISSION_STEP
        : uint16(performanceFee); // Compare with max commission step,
normalizing to MAX_COMMISSION_STEP

    performanceCommission = PerformanceFeeLib._getPerformanceCommission(
        packedPerformanceCommission,
        performanceCommission
    ); // Unpack commission percent for the step, where step is APY multiplier

    performanceFee = (profit * performanceCommission) /
PerformanceFeeLib.ONE_HUNDRED_PERCENT; // Profit * commission rate
}
```

Remediation:

The profit calculation should be corrected by removing the protocol fee and management fee, as follows:

```
if (totalCost - protocolFee - managementFee > highWaterMark) {
    uint256 profit = totalCost - protocolFee - managementFee - highWaterMark;
    ...
```

UFARM1-7 | Lack of Slippage Protection in UFarmPool Withdrawals and Deposits

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

contracts/main/contracts/pool/UFarmPool.sol::quexCallback()

Description:

When depositing to or withdrawing from the **UFarmPool** contract, users must submit a request and wait for the **quexCallback()** function - called by **quexCore** - to process it.

The **quexCallback()** function receives an updated pool valuation via **response.value**, representing the pool's total asset value at the time of processing. Since there may be a time gap between when the request is submitted and when it is fulfilled, the value of the pool's shares could fluctuate. This introduces the risk of users receiving significantly more or fewer assets or shares than they initially expected.

Example:
Suppose the value per share is $10 at time **T**, and Alice submits a request to burn 100 shares expecting to receive $1,000. However, if the protocol incurs a loss before her request is processed - causing the share value to drop to $5 - Alice would receive only $500 when her request is executed. This outcome is unexpected and may be perceived as unfair or unsafe by users.

```
sharesToMint = _mintSharesByQuote(investor, amountToInvest, _totalCost);

// Adjust the total cost and total deposit
_totalCost += amountToInvest;
totalDeposit += amountToInvest;
```

```
// Process the withdrawal
amountToWithdraw = _processWithdrawal(
    investor,
    sharesToBurn,
    _totalCost,
    withdrawalRequestHash,
    withdrawItem.bearerToken
);

if (investor != ufarmFund && amountToWithdraw != 0) {
    // Mark the request as used
    __usedWithdrawalsRequests[withdrawalRequestHash] = true;

    // Delete the request from the pending withdrawals
    delete pendingWithdrawalsRequests[withdrawalRequestHash];
}
```

Remediation:

Consider adding a **minOutputAmount** field to the **QueueItem** struct. This would allow users to specify the minimum acceptable amount they are willing to receive, enabling basic slippage protection and improving user trust in the protocol.

UFARM1-8 | USDC Blacklist Can Trigger DoS in quexCallback() Function

Fixed

Severity:	Probability:	Impact:
Medium	Rare	High

Path:

contracts/main/contracts/pool/UFarmPool.sol#L554

Description:

The **UFarmPool.quexCallback()** function is responsible for processing all pending deposit and withdrawal requests from users. It iterates through the **depositQueue[]** and **withdrawQueue[]** arrays, handling all requests within a single transaction. However, this design introduces a flaw: if any individual deposit or withdrawal reverts, the entire transaction fails, preventing all other requests from being processed.

This issue becomes particularly problematic in scenarios involving certain token behaviors. Consider a pool where the **valueToken** is USDC, and USDT is also accepted as a whitelisted token. In this setup, users are permitted to deposit and withdraw using either USDC or USDT.

USDC introduces a unique complication due to its blacklist mechanism. Addresses flagged by the USDC issuer cannot receive USDC transfers. An attacker who controls such a blacklisted address can exploit this behavior to disrupt the system:

1. The attacker deposits USDT into the pool to receive shares.
2. They later use those shares to initiate a withdrawal, which receives USDC as the **bearerToken**.
3. Since the attacker is blacklisted by USDC, the transfer in **_processWithdrawal()** fails on the line 554:

```
IERC20(bearerToken).safeTransfer(investor, burnedAssetsCost);
```

This transfer reverts, and because all deposit and withdrawal logic is processed in a single transaction via **quexCallback()**, **all other user requests in that batch are also reverted**, even if they were valid.

The same type of denial-of-service vector exists when dealing with ERC777 tokens. An attacker can implement a malicious tokensReceived hook that intentionally reverts, thereby sabotaging transfers to their address and causing the entire quexCallback() execution to fail.

In summary, the core issue lies in the lack of isolation in request processing - **a single failing operation can block the execution of all others**, creating a vector for targeted denial-of-service attacks using blacklisted tokens or malicious ERC777 hooks.

```
function _processWithdrawal(
    address investor,
    uint256 sharesToBurn,
    uint256 _totalcost,
    bytes32 withdrawalRequestHash,
    address bearerToken
) private keepWithdrawalHash(withdrawalRequestHash) returns (uint256
burnedAssetsCost) {
    uint256 _totalSupply = totalSupply();
    burnedAssetsCost = (_totalcost * sharesToBurn) / _totalSupply;

    if (IERC20(bearerToken).balanceOf(address(this)) >= burnedAssetsCost) {
        _burn(investor, sharesToBurn);

        /// @audit the following line will revert if the bearerToken = USDC and
investor is a blacklisted address
        IERC20(bearerToken).safeTransfer(investor, burnedAssetsCost);
        emit Withdraw(investor, bearerToken, burnedAssetsCost,
withdrawalRequestHash);

        highWaterMark -= highWaterMark > burnedAssetsCost ? burnedAssetsCost :
highWaterMark;

        emit WithdrawRequestExecuted(investor, sharesToBurn,
withdrawalRequestHash);
    } else {
        burnedAssetsCost = 0;
    }

    return burnedAssetsCost;
}
```

Remediation:

Consider using a try-catch block when transferring tokens to the investor in the **_processWithdrawal** function. If an error occurs, do not burn shares and return 0 instead.

UFARM1-10 | Fund creation DoS by front-running factory call

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

FundFactory.sol:createFund#L47-L49

Description:

The FundFactory exposes the **createFund** function, which uses a beacon proxy and **CREATE2** to deploy a new UFarmFund contract. This function is called from UFarmCore in the corresponding **createFund** function.

The **salt** value for the **CREATE2** opcode is the application ID, passed to the FundFactory from UFarmCore:

```
function createFund(
    address _fundManager,
    bytes32 _applicationId
)
    external override
    ownerOrHaveTwoPermissions(uint8(Permissions.UFarm.Member),
uint8(Permissions.UFarm.ApproveFundCreation))
    nonReentrant returns (address fund)
{
    uint256 nextFundId = _funds.length();
    fund = fundFactory.createFund(_fundManager, _applicationId);
    _funds.add(fund);
    emit FundCreated(_applicationId, nextFundId, fund);
}
```

And in UFarmFactory it uses the SafeOPS library:

```
function createFund(address _manager, bytes32 _salt) external onlyLinked
returns (address fund) {
    return SafeOPS._safeBeaconCreate2Deploy(fundImplBeacon, _salt,
_getInitFundCall(_manager));
}

function _safeBeaconCreate2Deploy(
 address _beacon,
```

```
 bytes32 _salt,
 bytes memory _initCall
) internal returns (address addr) {
 try new BeaconProxy{salt: _salt}(_beacon, _initCall) returns (BeaconProxy
beaconProxy) {
  return address(beaconProxy);
 } catch {
  revert BeaconProxyDeployFailed();
 }
}
```

The function reverts if the deployment fails, which would be the case if the contract already exists.

The **FundFactory.createFund** has no access control and as such, any attacker can deploy new UFarmFund contracts from the FundFactory. The newly deployed fund may not have been added to the UFarmCore fund whitelist, but the address and salt have now been taken.

By front-running application IDs in fund creation, an attacker can DoS fund creation, as the fund manager's call would now revert.

Remediation:

We would recommend to gate the **FundFactory.createFund** function to be callable only by the UFarmCore contract.

UFARM1-5 | Incorrect order of amounts emitted in the event of UnoswapV2Controller::delegatedAddLiquidity

Fixed

Severity:	Probability:	Impact:
Low	Unlikely	Low

Path:

contracts/main/contracts/controllers/UnoswapV2Controller.sol#L320

Description:

In the **UnoswapV2Controller::delegatedAddLiquidity()** function, **tokenA** and **tokenB** are never sorted. However, **amountA** and **amountB**, obtained from **quoteExactLiquidityAmounts()**, are sorted according to the pair's token order.

```
(uint256 amountA, uint256 amountB, address pair) =
thisController.quoteExactLiquidityAmounts(
    tokenA,
    tokenB,
    alArgs.amountADesired,
    alArgs.amountBDesired,
    alArgs.amountAMin,
    alArgs.amountBMin,
    alArgs.deadline
);

bool reversed = tokenA > tokenB;

IERC20(tokenA).safeTransfer(pair, reversed ? amountB : amountA);
IERC20(tokenB).safeTransfer(pair, reversed ? amountA : amountB);
```

Therefore, the event emitted by this function has **amountA** and **amountB** in the incorrect order, since it doesn't consider the reserve values to determine the correct token order.

```
emit LiquidityAddedUnoV2(tokenA, tokenB, amountA, amountB, liquidity, pair,
PROTOCOL());
```

Remediation:

The event emitted by this function should also consider the reserve values to reorder **amountA** and **amountB**.

hexens

x

UFARM DIGITAL