hexens x UFARM DIGITAL

Security Review Report for UFarm.Digital

June 2026

Table of Contents

1. About Hexens
2. Executive summary
3. Security Review Details
    - Security Review Lead
    - Scope
    - Changelog
4. Severity Structure
    - Severity characteristics
    - Issue symbolic codes
5. Findings Summary
6. Weaknesses
    - Unbounded Gas in ERC-1271 Signature Validation Enables Batch-Settlement Griefing
    - Deposit and Withdrawal Requests Lack Output Slippage Protection
    - CCIP transfers cannot dispatch when the fee token is also the bridged token
    - Zero-value cross-chain resends add a token transfer payload
    - Redundant validation in createVault
    - unwhitelist function should enforce descending indexes
    - Full or curator pause does not stop stored ERC-1271 signatures

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 audit covers the UFarm Digital EVM Contracts, an on-chain asset management protocol for creating, operating, and upgrading managed vaults across EVM-compatible networks.

The review was conducted over a two-week period and included a comprehensive analysis of all smart contracts in the repository.

During the assessment, we did not identify any critical or high-severity issues. We identified three medium-severity findings, one of which could allow an attacker to grief the batch settlement process by exhausting gas during ERC-1271 signature validation. In addition, we identified three informational findings.

All identified issues were either remediated or acknowledged by the development team and subsequently verified by our auditors.

Following the remediation phase, we conclude that the protocol's overall security posture and code quality have improved as a result of this audit.

3. Security Review Details

▪ Review Led by

Trung Dinh, Lead Security Researcher

▪ Scope

The analyzed resources are located on:

🔗 https://github.com/UFarmDigital/Contracts-EVM-v2-private/tree/f833b54078b04e0eea91f6c21cd241aefd60d114

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

🔗 https://github.com/UFarmDigital/UFarm-EVM-Contracts/commit/dbc50ba3930c6c257ecee1cc6dab51af20a739c6

▪ Changelog

29 June 2026	Audit start
15 July 2026	Initial report
20 July 2026	Revision received
22 July 2026	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.

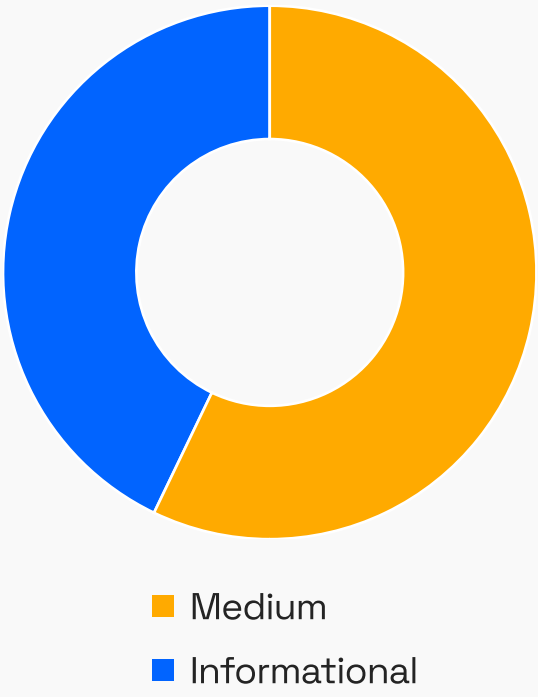
- **Issue Symbolic Codes**

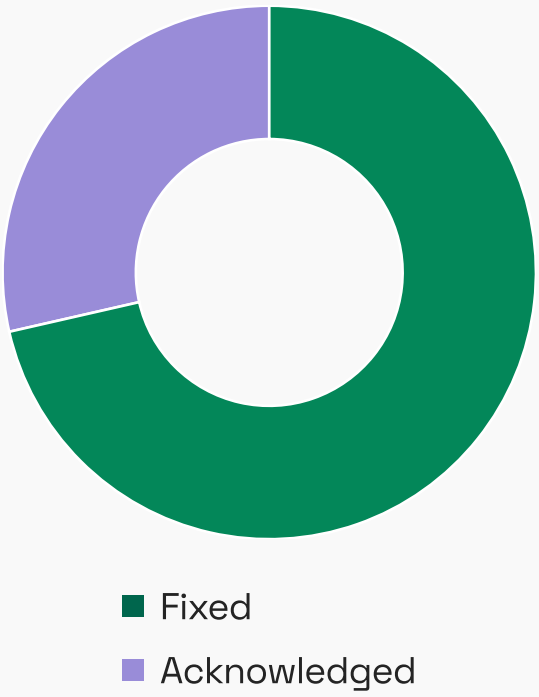
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

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

5. Findings Summary

Severity	Number of findings
Critical	0
High	0
Medium	4
Low	0
Informational	3
Total:	**7**





6. Weaknesses

This section contains the list of discovered weaknesses.

UFARM2-1 | Unbounded Gas in ERC-1271 Signature Validation Enables Batch-Settlement Griefing

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

contracts/vault/modules/VaultLifecycleModule.sol#L561-L565

Description:

Function **settleRequests(deposits, redeems)** processes a batch of signed deposit and redeem requests and is intended to skip invalid requests without reverting the entire batch. The **deposits** and **redeems** arrays are provided by privileged caller, we assume that they contain user-signed requests aggregated by some backend (BE).

For each request, **_validateSignedRequest()** calls **_isValidOwnerSignature()** to verify the owner's signature. When the owner is a contract, the function performs an external ERC-1271 call without a gas limit:

```
try IERC1271(owner_).isValidSignature(digest, signature) returns (bytes4 magicValue) {
    return magicValue == IERC1271.isValidSignature.selector;
} catch {
    return false;
}
```

Although the surrounding **try/catch** appears to make the call safe, it does not protect against gas griefing. Under EIP-150's the callee receives 63/64 of the remaining gas and can consume all of it (e.g., by entering an infinite loop). When the sub-call runs out of gas, execution enters the **catch** block and returns **false**, but only about 1/64 of the pre-call gas remains in the caller. This often leaves insufficient gas to continue processing the remaining requests or execute _startOracleRequest(), causing the entire settleRequests() transaction to revert with an out-of-gas error.

An attacker can exploit this by submitting a valid-looking signed request whose owner is a malicious ERC-1271 contract designed to consume all forwarded gas. If the backend includes

this request in a batch alongside legitimate requests, the entire settlement transaction fails. With two or more such requests in the same batch, the remaining gas shrinks to approximately **(1/64)²** of the original gas, making the transaction effectively impossible to complete regardless of the gas limit supplied by the caller.

The impact is a recoverable griefing DoS: settlement of the affected batch is blocked, and the caller's gas is wasted, defeating the intended design of skipping invalid requests rather than reverting the entire batch.

Remediation:

Consider capping the gas forwarded to the untrusted ERC-1271 call so that a malicious contract can consume only a bounded amount of gas. For example:

```
IERC1271(owner_).isValidSignature{gas: ERC1271_GAS_LIMIT}(digest, signature)
```

UFARM2-2 | Deposit and Withdrawal Requests Lack Output Slippage Protection

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

contracts/vault/modules/VaultLifecycleModule.sol#L309-L347

Description:

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

The **onOracleNav()** function receives an updated pool valuation via **response.value**, which represents the pool's total asset value at the time the request is executed.

Because there may be a delay between request submission and execution, the value of the vault shares can change. The protocol partially mitigates this by calling **_validatePrice()**, which ensures the share price does not move beyond a configured threshold.

```
function _validatePrice(
    uint256 nav,
    uint256 pricingSupply,
    VaultLifecycleStorage.Layout storage lifecycle
) internal returns (bool) {
    ...

    uint256 actualPriceFluctuation = Math.mulDiv(delta, WAD, previousPrice);
    if (actualPriceFluctuation <= allowedPriceFluctuation) {
        lifecycle.lastTokenPrice = newPrice;
        return true;
    }

    lifecycle.isInvalidPrice = true;
    emit InvalidPriceRaised(previousPrice, newPrice, allowedPriceFluctuation, actualPriceFluctuation);

    return false;
}
```

This guarantees that the share price at execution cannot deviate from the request-time price by more than the configured threshold.

However, this protection is incomplete because it only limits changes in the share price. It does not account for changes in the USD value of the deposited or withdrawn token (request.token) between request submission and execution.

For example:

- Assume VaultBaseStorage.data().asset = ETH.
- At time T_1:
  - Share price = $1,000
  - ETH price = $1,000
- Alice submits a request to deposit 100 ETH and expects to receive 100 shares.
- At time T_2:
  - ETH price drops to $500.
  - The share price remains $1,000 (so **_validatePrice()** passes).

When Alice's request is executed, the deposited assets are now worth only $50,000, so she receives:

```
50,000 / 1,000 = 50 shares
```

Instead of the 100 shares she expected when submitting the request, Alice receives only 50 shares, even though the share price validation succeeds.

Remediation:

Consider adding a **minOutput** parameter to deposit and withdrawal requests, allowing users to specify the minimum number of shares (for deposits) or assets (for withdrawals) they are willing to receive. The request should revert if the actual output falls below this value.

Commentary from the client:

"We developed our Vaults conforming to https://eips.ethereum.org/EIPS/eip-7540 which does not suppose slippage protections.
So we introduce the minAmount into the extended methods we added above the EIP, preserving the standard methods with no protection."

UFARM2-3 | CCIP transfers cannot dispatch when the fee token is also the bridged token

Acknowledged

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

contracts/adapters/CrossChainAdapter.sol#L476-L488

Description:

When **crossChainFeeToken** and **crossChainToken** are the same ERC-20 token, the function **_setRouterApprovals()** first approves the fee amount and then overwrites that allowance with the bridged token amount. As a result, a CCIP router that pulls both the fee and the bridged tokens never receives an allowance equal to **amount + fee**, causing any non-zero **TransferSent** dispatch to fail.

As a result, if a vault is configured to use the same token for both CCIP fees and bridged transfers, any non-zero cross-chain transfer will fail due to insufficient router allowance. The outbox remains active, settlement becomes blocked, and **manualResendCrossChainMessage()** cannot recover because it repeats the same approval logic. Recovery therefore requires either privileged reconfiguration to use a different fee token or a code fix.

```
function _setRouterApprovals(
    address feeToken,
    address asset,
    address router,
    CrossChainTypes.CrossChainMessageType messageType,
    uint256 feeAmount,
    uint256 assetAmount
) internal {
    IERC20(feeToken).forceApprove(router, feeAmount);
    if (messageType == CrossChainTypes.CrossChainMessageType.TransferSent) {
        IERC20(asset).forceApprove(router, assetAmount);
    }
}
```

Remediation:

Handle the same-token case explicitly. If **messageType == TransferSent** and **feeToken == asset**, approve **feeAmount + assetAmount** once, otherwise keep separate approvals. Clear the combined allowance after success or failure, and add a router mock test that pulls both fee token and bridged token amounts.

Commentary from the client:

“Intended implementation simplification. The designed constraint: bridge USDC only, pay fee with LINK only”

UFARM2-4 | Zero-value cross-chain resends add a token transfer payload

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

contracts/adapters/CrossChainAdapter.sol#L299-L302

Description:

Function **CrossChainAdapter.manualResendCrossChainMessage()** is used to resend the currently queued cross-chain outbox message. When the outbox message type is **TransferSent**, it always creates a **tokenAmounts** array containing one element for the CCIP dispatch.

```
if (outboxMessageType == CrossChainTypes.CrossChainMessageType.TransferSent) {
    crossChainToken = _requireCrossChainToken(crossChainToken);
    tokenAmounts = new CCIPClient.EVMTokenAmount[](1);
    tokenAmounts[0] = CCIPClient.EVMTokenAmount({token: crossChainToken, amount:
outbox.message.amount});
}
```

However, this behavior is inconsistent with **crossChainTransfer()**. When transferring **0** tokens from a satellite chain, **crossChainTransfer()** creates an empty **tokenAmounts** array rather than an array containing a zero-amount transfer.

```
if (isAccounting) {
    tokenAmounts = new CCIPClient.EVMTokenAmount[](1);
    tokenAmounts[0] = CCIPClient.EVMTokenAmount({token: crossChainToken, amount: amount});
}
```

Assume a zero-value **TransferSent** message fails to be dispatched by **crossChainTransfer()**. When the manager later calls **manualResendCrossChainMessage()** to resend the outbox, the resend path constructs a **tokenAmounts** array containing one element instead of the empty array used by the original dispatch. As a result, the resent message no longer matches the original one.

This inconsistency causes the transaction to revert on the destination chain because **_validateDeliveredTokens()** explicitly requires **destTokenAmounts.length == 0** when **message.amount == 0**.

```
function _validateDeliveredTokens(
    address asset,
    CrossChainTypes.CrossChainMessage memory message,
    CCIPClient.EVMTokenAmount[] calldata destTokenAmounts
) internal pure returns (bool, uint256) {
    if (message.messageType != CrossChainTypes.CrossChainMessageType.TransferSent) {
        return (destTokenAmounts.length == 0, 0);
    }

    if (message.amount == 0) {
        return (destTokenAmounts.length == 0, 0);
    }

    uint256 deliveredAmount = 0;
    for (uint256 i = 0; i < destTokenAmounts.length; ++i) {
        if (destTokenAmounts[i].token != asset) return (false, 0);
        deliveredAmount += destTokenAmounts[i].amount;
    }
    if (deliveredAmount == 0) return (false, 0);
    return (true, deliveredAmount);
}
```

As shown in lines 556–558, the transaction reverts whenever **message.amount == 0** while **destTokenAmounts.length != 0**. Consequently, a failed zero-value **TransferSent** message cannot be successfully resent using **manualResendCrossChainMessage()**.

Remediation:

Make resend reconstruction match the original dispatch rules. In **manualResendCrossChainMessage**, attach a CCIP token amount only when the outbox message is **TransferSent** and **amount != 0**.

UFARM2-5 | Redundant validation in createVault

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

contracts/curator/UFarmCurator.sol#L144-L145
contracts/curator/UFarmCurator.sol#L148

Description:

In **UFarmCurator.createVault()**, the function reverts if **vaultOwner** or **asset** is **address(0)**, if **name** or **symbol** is empty, or if the asset has a zero USD value.

```
if (vaultOwner == address(0) || asset == address(0)) revert InvalidAddress();
if (bytes(name).length == 0 || bytes(symbol).length == 0) revert InvalidMetadata();
...
if (IUFarmCore(core_).valueTokenToUsd(asset_, WAD) == 0) revert IUFarmVaultErrors.InvalidAsset();
```

The function then deploys a new vault and calls its **initialize()** function. However, **UFarmVault.initialize()** performs the same validations.

```
if (core_ == address(0) || curator_ == address(0) || owner_ == address(0) || asset_ == address(0)) {
    revert IUFarmVaultErrors.InvalidAddress();
}
if (bytes(name_).length == 0 || bytes(symbol_).length == 0) revert IUFarmVaultErrors.InvalidMetadata();
if (coreContract.valueTokenToUsd(asset, WAD) == 0) revert InvalidAsset();
```

As a result, the zero-address, empty-metadata, and asset-value checks in **UFarmCurator.createVault()** are redundant because they are already enforced by **UFarmVault.initialize()**.

Remediation:

Consider removing the duplicate validation logic from **UFarmCurator.createVault()** and relying on **UFarmVault.initialize()** to validate the input parameters.

UFARM2-6 | unwhitelist function should enforce descending indexes

Acknowledged

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

contracts/adapters/ArbitraryAdapterGuard.sol#L103-L129
contracts/adapters/ArbitraryAdapterGuard.sol#L162-L187

Description:

Function **ArbitraryAdapterGuard.unwhitelistProtocol()** is used to remove allowed protocol call directives. The function removes entries with swap-and-pop technique hence when removing multiple entries, we should pass unique indexes in descending order so earlier removals do not invalidate later indexes. This is already noticed in the **@dev** tag comment of the function.

```
/*
 * @dev Removes entries with swap-and-pop. When removing multiple entries, pass unique indexes in descending order so
 *      earlier removals do not invalidate later indexes. Duplicate indexes can remove entries different from the
 *      caller's intended original positions.
 */
function unwhitelistProtocol(bytes32 dapp, address target, uint256[] calldata indexes)
        external
        onlyCoreAdapterManager
    {
        Directive[] storage dirs = whitelist[dapp][target];
        if (indexes.length == 0) revert ArbitraryAdapterGuardInvalidInput();

        for (uint256 i = 0; i < indexes.length; ++i) {
            uint256 idx = indexes[i];
            if (idx >= dirs.length) revert ArbitraryAdapterGuardIndexOutOfBounds();

            bytes4 method = _extractSelector(dirs[idx].directives);
            dirs[idx] = dirs[dirs.length - 1];
            dirs.pop();

            emit ArbitraryAdapterWhitelistUpdated(dapp, target, method, false);
        }
    }
```

However this is not enforced why implementing the funciton. As the consequence, there is a chance that the manager could pass the indexes in arbitrary order which could result in incorrect entries removal.

Note that a similar issue happens in function **ArbitraryAdapterGuard.unwhitelistEIP712()**.

Remediation:

Consider validating that the supplied indexes are unique and strictly sorted in descending order before performing any removals.

Commentary from the client:

“Noticed at natspec, could be enforced on backend/frontend side”

UFARM2-7 | Full or curator pause does not stop stored ERC-1271 signatures

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

contracts/vault/UFarmVault.sol#L551-L561

Description:

The **signMessage()** function is disabled when either the full protocol pause or the curator pause is active. However, **isValidSignature()** does not enforce the same pause checks. As a result, a signature authorized before an emergency pause remains valid and can still be consumed by external protocols through the ERC-1271 interface.

This allows vault-controlled assets to continue being moved, approved, or otherwise encumbered via ERC-1271 integrations, even though operators may expect a full or curator pause to halt all downstream protocol activity.

```
function isValidSignature(bytes32 hash, bytes memory signature) external view returns (bytes4 magicValue) {
    if (VaultLifecycleStorage.data().activeOracleRequest.requestId != 0) return
ERC1271_INVALID_SIGNATURE;

    if (!VaultPermissionStorage.data().signedMessages[hash]) return ERC1271_INVALID_SIGNATURE;

    // slither-disable-next-line unused-return
    (address signer, ECDSA.RecoverError recoverError,) = ECDSA.tryRecover(hash, signature);
    if (recoverError != ECDSA.RecoverError.NoError) return ERC1271_INVALID_SIGNATURE;
    if (!_hasPermission(signer, MEMBER | VAULT_ASSET_MANAGER)) return
ERC1271_INVALID_SIGNATURE;

    return IERC1271.isValidSignature.selector;
}
```

Remediation:

Make **isValidSignature()** fail closed while emergency pause modes are active. At a minimum, return an invalid signature when **FULL_PROTOCOL_PAUSE** is set or when the curator pause mask is non-zero. Consider also honoring **ADAPTER_EXECUTION_PAUSED** so that ERC-1271 signature validation and adapter execution share the same emergency pause boundary.

hexens

x

UFARM DIGITAL