hexens × Spool

AUG.24

SECURITY REVIEW REPORT FOR SPOOL

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.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered the MetaVault smart contracts, as developed by Yelay protocol. The MetaVault sits between the user and the SmartVaults and allows for easy managing of positions in multiple SmartVaults, as well as tokenizing these positions in MVT.

Our security assessment was a full review of the smart contracts, spanning a total of 2 weeks.

During our audit, we have identified 1 high severity vulnerability, which could have lead to potential failure of the MetaVault.

We have also identified several minor severity vulnerabilities and code optimisations.

A critical vulnerability was also discovered by the Yelay team after the audit, which has been included in this report with our analysis and remediation.

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 have increased after completion of our audit.

SCOPE

The analyzed resources are located on:

https://github.com/solidant/spool-v2-core/tree/3e29af3fe8d306177b639cf6b2e97ddc2c2ce2e5

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

https://github.com/solidant/spool-v2-core/commit/fda7ae775a3e584428a1194d3d3f4fa67409c06c

https://github.com/solidant/spool-v2-core/tree/fe413be12423d0f98baab6e16f38da059e955e4d

AUDITING DETAILS

STARTED
26.08.2024

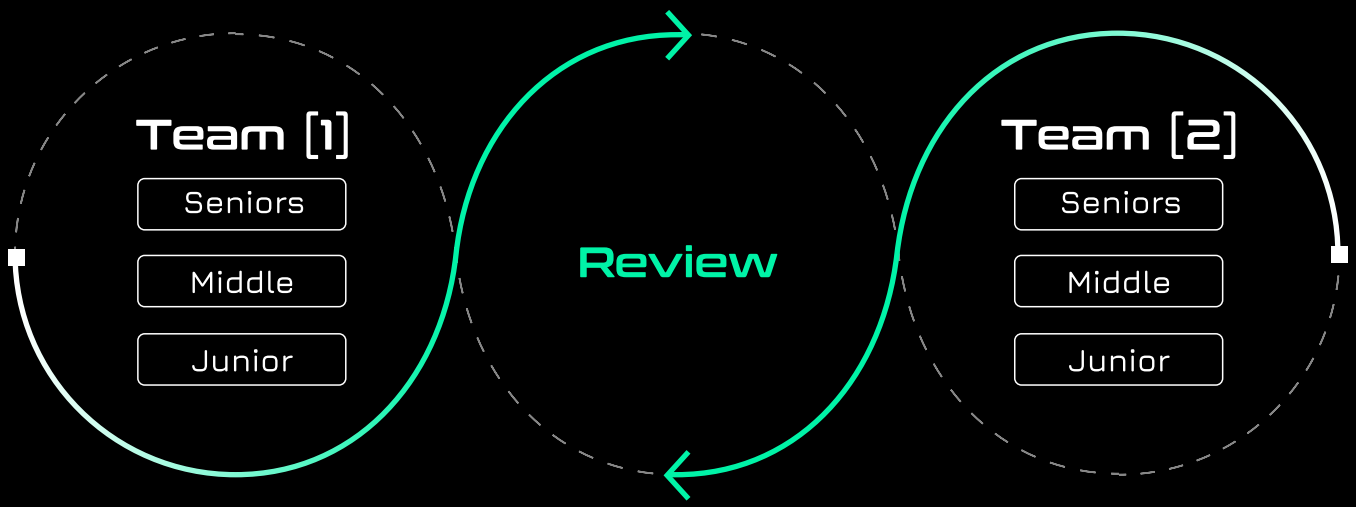
DELIVERED
02.09.2024

Review Led by

NOUREDDINE BENOMARI
Security Researcher | Hexens

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

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 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.

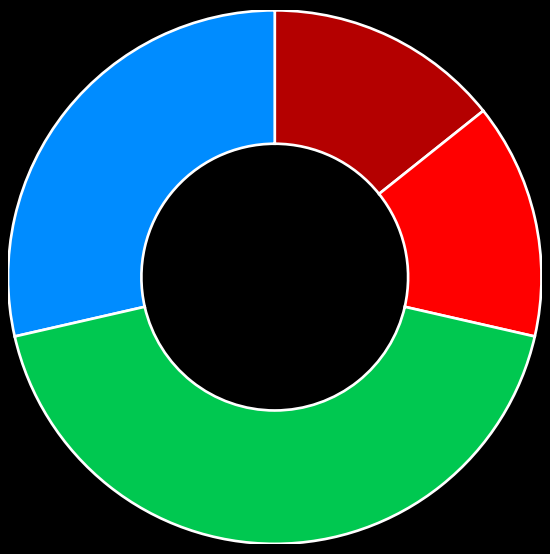
ISSUE SYMBOLIC CODES

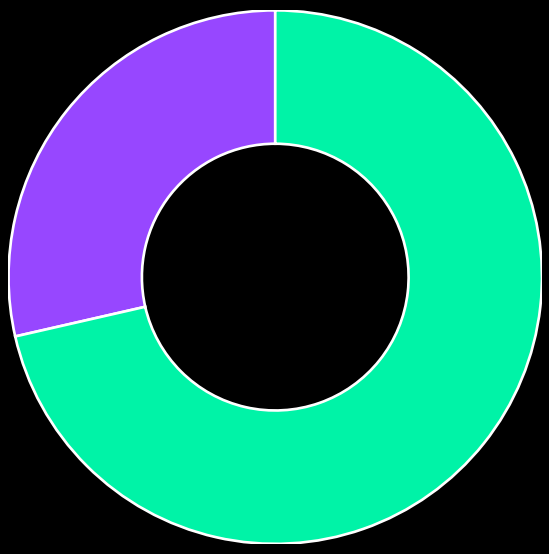
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	1
High	1
Medium	0
Low	3
Informational	2

Total: 7





WEAKNESSES

This section contains the list of discovered weaknesses.

SPOOL7-7

METAVAULT TOTAL SUPPLY MANIPULATION CAN LEAD TO ASSET THEFT

SEVERITY: Critical

PATH:

src/MetaVault.sol:_syncDeposit#L458

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The MetaVault uses a flush-sync mechanism, where a user's deposit is first deposited into a Spool SmartVault during the flush and later claimed as SVT in the sync. It is in the sync that the user will know exactly how many shares of the MetaVault it will get. Once the sync is finished, the user needs to claim their tokens.

This share rate is dependent on the sync index and can be different each round. On line 469 we see that the calculation is a simple share rate calculation using the total balance, deposited assets and total supply:

```
toMint = (totalSupply_ * depositedAssets) / (totalBalance - depositedAssets)
```

The **totalSupply_** is the total supply at the moment of sync, fetched on line 458.

The issue here is that the total supply can change, while a user can have a pending withdrawal with a fixed share amount before the sync. This can be seen in the **redeem** function:

```
function redeem(uint256 shares) external {
    _checkNotPaused();
    _burn(msg.sender, shares);
    uint128 flushIndex = index.flush;
    // accumulate redeems for all users for current flush index
    flushToRedeemedShares[flushIndex] += shares;
    // accumulate redeems for particular user for current flush index
    userToFlushToRedeemedShares[msg.sender][flushIndex] += shares;
    emit Redeem(msg.sender, flushIndex, shares);
}
```

The user's shares are burned immediately, which means the total supply is lowered, while shares are counted in the user's redeemed shares for this flush index. This wouldn't be an issue if the withdrawals were done immediately, since they would have gotten their fair share of the total balance amount.

But in the **sync** function, we see that deposits are processed before withdrawals:

```
function sync() external {
    _checkNotPaused();
    _checkOperator();
    address[] memory vaults = _smartVaults.list;
    Index memory index_ = index;
    if (vaults.length > 0 && index_.sync < index_.flush) {
        (bool depositHadEffect, uint256 totalBalance) = _syncDeposit(vaults,
index_.sync);
```

```
            bool withdrawalHadEffect = _syncWithdrawal(vaults, index_.sync);
            if (depositHadEffect || withdrawalHadEffect) {
                index.sync++;
                emit SharePrice(totalBalance, totalSupply());
            }
        }
    }
```

The total supply is lowered and so the **toMint** calculation will also result in fewer shares for the other users. The withdrawal of the original user still has the full amount of shares and as a result they would be taking a larger portion of the total balance.

Consider the following scenario:

- User A holds 100 MVT and the MetaVault has 100 ETH balance.
- User B deposits 100 ETH.
- The MetaVault is flushed, making **depositedAssets** 100 ETH for this index.
- User A redeems 99 MVT, total supply is now 1 MVT.
- The MetaVault is synced, making **totalBalance** 200 ETH and **toMint = (1 * 100) / (200 - 100) = 1 MVT**.
- User B now claims their 1 MVT.
- The MetaVault is flushed, which will redeem **99 / 101** from the SmartVault.
- The MetaVault is synced, granting 196 ETH to user A.
- User A claims 196 ETH, leaving user B with 1 MVT worth 2 ETH.

This attack scenario can be repeated on every flush-sync round, not just on empty vaults. It allows for user asset theft.

```
function _syncDeposit(address[] memory vaults, uint128 syncIndex)
    internal
    returns (bool hadEffect, uint256 totalBalance)
{
    uint256 depositedAssets = flushToDepositedAssets[syncIndex];
    if (depositedAssets > 0) {
        for (uint256 i; i < vaults.length; i++) {
            uint256[] memory depositNfts = new uint256[](1);
            depositNfts[0] = smartVaultToDepositNftId[vaults[i]];
            if (depositNfts[0] > 0) {
                uint256[] memory nftAmounts = new uint256[](1);
                nftAmounts[0] =
ISmartVault(vaults[i]).balanceOfFractional(address(this), depositNfts[0]);
                // make sure there is actual balance for given nft id
                if (nftAmounts[0] == 0) revert NoDepositNft(depositNfts[0]);
                smartVaultManager.claimSmartVaultTokens(vaults[i], depositNfts,
nftAmounts);
                delete smartVaultToDepositNftId[vaults[i]];
                hadEffect = true;
            }
        }
    }
    (totalBalance,) = _getBalances(vaults);
    if (hadEffect) {
        uint256 totalSupply_ = totalSupply();
        uint256 toMint;
        if (totalSupply_ < sharesToLock) {
            toMint = depositedAssets * INITIAL_SHARE_MULTIPLIER;
            uint256 lockedSharesLeftToMint = sharesToLock - totalSupply_;
            if (toMint < lockedSharesLeftToMint) {
                lockedSharesLeftToMint = toMint;
            }
            toMint -= lockedSharesLeftToMint;
            _mint(INITIAL_LOCKED_SHARES_ADDRESS, lockedSharesLeftToMint);
        } else {
            toMint = (totalSupply_ * depositedAssets) / (totalBalance -
depositedAssets);
        }
        flushToMintedShares[syncIndex] = toMint;
        _mint(address(this), toMint);
        emit SyncDeposit(syncIndex, toMint);
    }
}
```

During a flush, the total supply should be recorded for that flush index, similar to how the deposited assets are finalized in **flushToDepositedAssets** in a flush due to the incrementing of the flush index.

This cached total supply should be used to calculate the **toMint** value of the deposited assets.

For example:

```
function flush() external {
    _checkNotPaused();
    _checkOperator();
    _checkPendingSync();
    if (needReallocation) revert NeedReallocation();

    address[] memory vaults = _smartVaults.list;
    if (vaults.length > 0) {
        uint128 flushIndex = index.flush;
        // we process withdrawal first to ensure all SVTs are collected
        bool withdrawalHadEffect = _flushWithdrawal(vaults, flushIndex);
        bool depositHadEffect = _flushDeposit(vaults, flushIndex);
        if (withdrawalHadEffect || depositHadEffect) {
+           flushToTotalSupply[flushIndex] = totalSupply();
            index.flush++;
        }
    }
}
```

Note: The issue was found by the Yelay internal team, and validated by Hexens.

METAVAULT DOS BY USER THROUGH SYNC MECHANISM

SEVERITY: High

PATH:

src/MetaVault.sol:_syncDeposit#L469

REMEDIATION:

The edge case where depositedAssets == totalBalance should be handled separately as the share is technically infinite (division by zero). It should use a simple 1:1 share rate in that case, like other ERC4626 standard vaults.

STATUS: Fixed

DESCRIPTION:

The MetaVault uses a flush-sync mechanism to handle user deposits/withdrawals and spool smart vault deposits/withdrawals. It is important that a flush is always followed by a sync, however a user is able to DoS this mechanism in some cases, leading to potentially stuck assets.

The function **_syncDeposit** is used to claim the smart vault tokens from the manager and calculate the current total balance of the MetaVault. This balance is then used to calculate the amount of shares to-be-minted to the new depositors using the deposited amount. This calculation can lead to DoS through revert due to division-by-zero on line 469:

```
toMint = (totalSupply_ * depositedAssets) / (totalBalance -
depositedAssets);
```

This line is part of the else branch of a conditional, where the total supply had been initialised by some initial locked shares (to combat inflation attacks). The calculation **totalBalance - depositedAssets** can equal zero if the MetaVault was empty at some point in time causing the next **sync** to trigger this bug. The execution will revert and it will become impossible to call sync on the MetaVault, leading to DoS.

This bug can also be forced if the MetaVault is new and so an attacker can target new MetaVaults for more success. It can also lead to user-owned assets getting stuck if those assets are in a withdrawal request. The vulnerable sync would process the withdrawals first and so the balance of the vault is technically zero plus the attackers fresh deposit, triggering the bug.

```
    function _syncDeposit(address[] memory vaults, uint128 syncIndex)
        internal
        returns (bool hadEffect, uint256 totalBalance)
    {
        uint256 depositedAssets = flushToDepositedAssets[syncIndex];
        if (depositedAssets > 0) {
            for (uint256 i; i < vaults.length; i++) {
                uint256[] memory depositNfts = new uint256[](1);
                depositNfts[0] = smartVaultToDepositNftId[vaults[i]];
                if (depositNfts[0] > 0) {
                    uint256[] memory nftAmounts = new uint256[](1);
                    nftAmounts[0] =
ISmartVault(vaults[i]).balanceOfFractional(address(this), depositNfts[0]);
                    // make sure there is actual balance for given nft id
                    if (nftAmounts[0] == 0) revert NoDepositNft(depositNfts[0]);
                    smartVaultManager.claimSmartVaultTokens(vaults[i], depositNfts,
nftAmounts);
                    delete smartVaultToDepositNftId[vaults[i]];
                    hadEffect = true;
                }
            }
        }
        (totalBalance,) = _getBalances(vaults);
        if (hadEffect) {
            uint256 totalSupply_ = totalSupply();
            uint256 toMint;
            if (totalSupply_ < sharesToLock) {
                toMint = depositedAssets * INITIAL_SHARE_MULTIPLIER;
                uint256 lockedSharesLeftToMint = sharesToLock - totalSupply_;
                if (toMint < lockedSharesLeftToMint) {
                    lockedSharesLeftToMint = toMint;
                }
                toMint -= lockedSharesLeftToMint;
                _mint(INITIAL_LOCKED_SHARES_ADDRESS, lockedSharesLeftToMint);
            } else {
                toMint = (totalSupply_ * depositedAssets) / (totalBalance -
depositedAssets);
            }
            flushToMintedShares[syncIndex] = toMint;
            _mint(address(this), toMint);
            emit SyncDeposit(syncIndex, toMint);
        }
    }
```

MISSING _DISABLEINITIALIZERS() IN METAVAULT IMPLEMENTATION CONTRACT

SEVERITY: Low

PATH:

src/MetaVault.sol#L135-L148

REMEDIATION:

It is recommended to add a call to _disableInitializers() within the constructor of the MetaVault implementation contract. This will ensure that the implementation contract cannot be initialized, protecting it from unauthorized control or misuse.

STATUS: Fixed

DESCRIPTION:

The **MetaVault** contract does not include **_disableInitializers()** in its constructor. This omission leaves the implementation contract vulnerable to unauthorized or accidental initialization, which could potentially allow an attacker to take control of the implementation contract.

```
 constructor(
        ISmartVaultManager smartVaultManager_,
        ISpoolAccessControl spoolAccessControl_,
        IMetaVaultGuard metaVaultGuard_,
        ISpoolLens spoolLens_
    ) SpoolAccessControllable(spoolAccessControl_) {
        if (
            address(smartVaultManager_) == address(0) || address(metaVaultGuard_) ==
address(0)
                || address(spoolLens_) == address(0)
        ) revert ConfigurationAddressZero();
        smartVaultManager = smartVaultManager_;
        metaVaultGuard = metaVaultGuard_;
        spoolLens = spoolLens_;
    }
```

VALIDATESMARTVAULTS FUNCTION RETURNS TRUE WHEN SMARTVAULTS ARRAY IS EMPTY

SEVERITY: Low

PATH:

src/MetaVaultGuard.sol#L47-L52

REMEDIATION:

Consider explicitly handling the case where smartVaults.length == 0.

STATUS: Fixed

DESCRIPTION:

In **MetaVaultGuard** The **validateSmartVaults** function **returns** true even when the **smartVaults** array is empty, as the loop does not execute and the function proceeds to return **true** by default. Although this behavior is not exploitable in the current implementation, it may lead to unexpected results if this function is used in future implementations or by other projects where an empty array should not be considered valid.

```
function validateSmartVaults(address asset, address[] calldata smartVaults)
external view virtual returns (bool) {
        for (uint256 i; i < smartVaults.length; i++) {
            _validateSmartVault(asset, smartVaults[i]);
        }
        return true;
    }
```

POTENTIAL RISK IN PAUSING MECHANISM FOR DEPOSIT FUNCTIONS

SEVERITY: Low

PATH:

src/MetaVault.sol#L255-L257
src/MetaVault.sol#L260-L262
src/MetaVault.sol#L264-L272

REMEDIATION:

Ensure that the pausing mechanism explicitly accounts for both msg.sig values associated with the deposit functions. This will ensure that both versions of the function are treated consistently.

STATUS: Acknowledged

DESCRIPTION:

MetaVault uses a mechanism to pause functions based on their **msg.sig**, which is the function selector (the first 4 bytes of the Keccak-256 hash of the function signature).

The contract contains two versions of the **deposit** function:

- **deposit(uint256)**
- **deposit(address,uint256)**

Each function generates a unique function selector (**msg.sig**), which is derived from the first 4 bytes of the Keccak-256 hash of the function signature:

- **deposit(uint256)** has a **msg.sig** of **a5df5779**.
- **deposit(address,uint256)** has a **msg.sig** of **0d4fa9df**.

Issue:
Since the contract's pausing mechanism logic relies on **msg.sig** to identify the **deposit** function, there is a risk that only one version of the function will be paused or restricted while the other remains active. This could result in:

- **Unintended Functionality**: One version of the **deposit** function might not be paused as expected, allowing deposits to continue when they should be halted.
- **Security Risk**: In emergency scenarios where the contract's **deposit** functionality needs to be disabled, failure to pause both function versions could expose the contract to risks.

```
function deposit(uint256 amount) external {
        _deposit(amount, msg.sender);
    }
/// @inheritdoc IMetaVault
function deposit(uint256 amount, address receiver) external {
    _deposit(amount, receiver);
}

function _deposit(uint256 amount, address receiver) internal {
    _checkNotPaused();
    uint128 flushIndex = index.flush;
    // MetaVault has now more funds to manage
    flushToDepositedAssets[flushIndex] += amount;
    userToFlushToDepositedAssets[receiver][flushIndex] += amount;
    IERC20MetadataUpgradeable(asset).safeTransferFrom(msg.sender,
address(this), amount);
    emit Deposit(receiver, flushIndex, amount);
}
```

SPOOLMULTICALL.AGGREGATE3() DOESN'T SPECIFY GAS FOR THE CALLS AND CAN BE DOSED

SEVERITY: Informational

PATH:

src/SpoolMulticall.sol#L41-L72

REMEDIATION:

We recommend adding an option to specify the gas limit for calls or a warning.

STATUS: Acknowledged

DESCRIPTION:

It is possible to DoS the **SpoolMulticall.aggregate3()** function on a call to an untrusted address because there's no mechanism to limit the gas for calls. The call will always receive 63/64 of the gas, and if used, the remaining 1/64 will not be sufficient to cover the costs of any subsequent loop executions, so the execution will halt.

```
    /// @notice Aggregate calls, ensuring each returns success if required
    /// @param calls An array of Call3 structs
    /// @return returnData An array of Result structs
    function aggregate3(Call3[] calldata calls) public payable returns
(Result[] memory returnData) {
        if (tx.origin != address(0)) _checkRole(ROLE_SPOOL_MULTI_CALLER,
msg.sender);
        uint256 length = calls.length;
        returnData = new Result[](length);
        Call3 calldata calli;
        for (uint256 i = 0; i < length;) {
            Result memory result = returnData[i];
            calli = calls[i];
            (result.success, result.returnData) =
calli.target.call(calli.callData); // @audit-issue still possibile to revert
even if failure is allowed by using up the gas (and not possible by
returning a memory bomb because mload will only load 1 word). This function
is not used anywhere, I think, so info
            assembly {
                // Revert if the call fails and failure is not allowed
                // `allowFailure := calldataload(add(calli, 0x20))` and
`success := mload(result)`
                if iszero(or(calldataload(add(calli, 0x20)), mload(result)))
{
                    // set "Error(string)" signature:
bytes32(bytes4(keccak256("Error(string)")))
                    mstore(0x00,
0x08c379a000000000000000000000000000000000000000000000000000000000)
                    // set data offset
                    mstore(0x04,
0x0000000000000000000000000000000000000000000000000000000000000020)
                    // set length of revert string
                    mstore(0x24,
0x0000000000000000000000000000000000000000000000000000000000000017)
                    // set revert string:
bytes32(abi.encodePacked("Multicall3: call failed"))
                    mstore(0x44,
0x4d756c746963616c6c333a2063616c6c206661696c656400000000000000000000)
                    revert(0x00, 0x64)
                }
            }
            unchecked {
                ++i;
            }
        }
    }
```

SPOOL7-3

MULTICALLS WITH PERMITASSET() AND PERMITDAI() CAN BE DOSED

SEVERITY: Informational

PATH:

src/MetaVault.sol#L712-L723

REMEDIATION:

We recommend making the permit calls in the two functions optional with the try-catch pattern like this:

try IERC20PermitUpgradeable(asset).permit(msg.sender, address(this), amount, deadline, v, r, s); {} catch {}

STATUS: Fixed

DESCRIPTION:

It is possible to DoS any multicall containing the permit functions, **permitAsset()** and **permitDai()**.

When such a multicall happens, attackers can extract the signature and frontrun the call with another transaction to the original asset's permit function, making the signature invalid because of the used nonce.

```
    /// @inheritdoc IMetaVault
    function permitAsset(uint256 amount, uint256 deadline, uint8 v, bytes32
r, bytes32 s) external { // @audit-issue if used as adviced with multicall,
can be DoSed by frontrunning
        _checkNotPaused();
        IERC20PermitUpgradeable(asset).permit(msg.sender, address(this),
amount, deadline, v, r, s);
    }

    /// @inheritdoc IMetaVault
    function permitDai(uint256 nonce, uint256 deadline, bool allowed, uint8
v, bytes32 r, bytes32 s) external {
        _checkNotPaused();
        IDAI(asset).permit(msg.sender, address(this), nonce, deadline,
allowed, v, r, s);
    }
```


hexens
×
Spool