Ommo Review

A smart contract assessment of Ommo contracts was performed by Octane Security, with a focus on the security aspects of the application's implementation.

About Octane Security

Octane Security is an AI-enhanced blockchain security firm dedicated to safeguarding digital assets. We combine security research with cutting-edge machine learning models to maximize protocol security and minimize risk. Our team ranks in the top of audit contest leaderboards, captures bug bounties, and has served big-name clients such as Alchemy, Optimism, Notional Finance, Blast, and Graph Protocol in the past.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We also rely on information that is provided by the client, its affiliates and partners for vulnerability identification. We can not guarantee the absense of vulnerabilities, issues, flaws, or defects after the review. It is up to the client to decide whether to implement recommendations and we take no liability for invalid recommendations. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

Security Assessment Summary

The scope of this review was limited to files at commit: b662357154496c83efc44fba1a0326cfcecd60bb

Scope

The following smart contract was in scope of the audit:

Smart Contract Scope

src/adapters/UniversalAdapterEscrowFactory.sol

src/adapters/UniversalAdapterEscrow.sol

src/valuers/UniversalValuerOffchain.sol

src/wrapper/UniversalTokenWrapper.sol

Off-Chain Script

src/keepers/OffchainValuationKeeper.py

Findings

Finding Number	Finding Name
H-1	Potential Double-Counting Of Escrow Token Balances In Strategy Valuation
M-1	First Mint Fails Due To Virtual Shares Miscalculation
L-1	Inconsistent Enforcement Of `MAX_PRICE_CHANGE_BPS` In Price Bound Logic
L-2	Lack Of Upper Bound Check On Nonce In `batchUpdateValues` And `updateValue`
L-3	Missing Validation Between `minUpdateInterval` And `maxStaleness`
L-4	Inaccurate Idle Asset Calculation In `getIdleAssets`
I-1	Lack Of Differentiation Or Logging Between Value Validity Scenarios In `getTotalValue`
I-2	Inconsistent Logic Between Confidence Threshold And Update Interval
I-3	Redundant Balance Retrieval In `deallocate` Function
I-4	Unused Variables: `dailyLimit` And `dailyUsed`

Title: H-1 Potential Double-Counting Of Escrow Token Balances In Strategy Valuation

Description

When updating a value in `underlying_balance` mode, the `_read_underlying_balance` function is called to retrieve the balance of a specific token held by the escrow. As a result, the reported value of a strategy is directly derived from the escrow's token holdings:

```
def _read_underlying_balance(self, token: str, holder: str) -> int:
    """Read ERC20 balanceOf(holder) for token."""
    erc, _ = self._erc20(token)
    try:
        return int(erc.functions.balanceOf(holder).call())
    except Exception as e:
        raise RuntimeError(f"balanceOf({token}, {holder}) failed: {e}")
```

Similarly, in `holdings` mode, the escrow balance is also queried directly, so the strategy's reported value is again based on the escrow's token holdings:

```
def value_holdings_mode(self, s: StrategyConfig) -> int:
    """
    Mode: 'holdings'
    - Aggregate all holdings with their respective signs (+1 for assets, -1 for
liabilities)
```

```
    - Supports multiple token types (ERC20, PT, etc.)
    - Converts total underlying to wrapper shares
    """
    try:
        holdings = s.extras.get('holdings', []) if s.extras else []
        total_underlying = 0

        for holding in holdings:
            holding_type = holding.get('type', 'erc20')
            token_addr = holding.get('token')
            sign = holding.get('sign', 1)

            if not token_addr:
                continue

            # Get token balance
            token, _ = self._erc20(token_addr)
            balance = int(token.functions.balanceOf(s.escrow).call())

            # Apply sign
            total_underlying += sign * balance

            logger.debug(f"[{s.id_text}] holding: token={token_addr}, balance=
{balance}, sign={sign}")

        # Ensure non-negative
        if total_underlying < 0:
            total_underlying = 0

        # Convert to wrapper shares
        shares = self._convert_underlying_to_wrapper_shares(total_underlying)
        logger.debug(f"[{s.id_text}] holdings total: underlying=
{total_underlying}, shares={shares}")
        return shares

    except Exception as e:
        logger.error(f"Error in holdings valuation: {e}")
        return 0
```

Later, when the `realAssets` function is called, it calls `getTotalValue`, where `report.value` for each strategy is aggregated into `totalValue`. However, at the end, the escrow's token balance is added again:

```
function getTotalValue(address escrow) external view override returns (uint256
totalValue) {
    // ...

    for (uint256 i = 0; i < strategies.length; i++) {
        bytes32 strategyId = strategies[i];
        ValueReport memory report = latestReports[strategyId];

        if (stalenessAge <= maxStaleness && report.confidence >= minConfidence) {
            totalValue += report.value;
        } else if (stalenessAge <= ABSOLUTE_MAX_STALENESS && report.confidence >=
minConfidence) {
            totalValue += report.value;
        } else if (fallbackValues[strategyId] > 0) {
            totalValue += fallbackValues[strategyId];
        } else if (report.value > 0 && stalenessAge <= ABSOLUTE_MAX_STALENESS) {
            totalValue += report.value;
        }
    }

    // Add idle assets
    totalValue += IERC20(asset).balanceOf(escrow);

    return totalValue;
}
```

This approach can lead to **double counting**, since `report.value` already includes the escrow's balance (fetched via `_read_underlying_balance`), and the same balance is added again at the end of `getTotalValue`.

Recommendation

It is recommended to either adjust the Python script to avoid including escrow balances twice when using these modes, or skip reading the balance for idle assets in `getTotalValue`.

Title: M-1 First Mint Fails Due To Virtual Shares Miscalculation

Description

There is a flaw in the `mint` function that can render it non-functional. Specifically, when `_totalSupply == 0 || beforeBal == 0`, the function enforces that `shares` must be less than or equal to `maxShares - VIRTUAL_SHARES` in order to account for the virtual shares:

```
function mint(uint256 shares, address receiver) external nonReentrant returns
(uint256 assets) {
    require(shares != 0, "WRP: zero shares");
    require(receiver != address(0), "WRP: recv=0");

    uint256 _totalSupply = totalSupply;
    uint256 beforeBal = IERC20(underlying).balanceOf(address(this));

    assets = previewMint(shares);
    SafeERC20Lib.safeTransferFrom(underlying, msg.sender, address(this), assets);
    uint256 received = IERC20(underlying).balanceOf(address(this)) - beforeBal;

    uint256 maxShares;
    if (_totalSupply == 0 || beforeBal == 0) {
        maxShares = received;
        require(maxShares > VIRTUAL_SHARES, "WRP: first mint too small");
        require(shares <= maxShares - VIRTUAL_SHARES, "WRP: insufficient recv");

        _mint(DEAD_ADDRESS, VIRTUAL_SHARES);
        _mint(receiver, shares);

        emit Deposit(msg.sender, receiver, received, shares);
        return assets;
    } else {
        maxShares = received.mulDivDown(_totalSupply, beforeBal);
        require(maxShares >= shares, "WRP: insufficient recv");
        _mint(receiver, shares);
        emit Deposit(msg.sender, receiver, received, shares);
        return assets;
    }
}
```

The problem arises in the first mint scenario. For example, if a user wants to mint 1100 shares by calling `mint(1100, user)`, the function executes as follows:

1. `previewMint(1100)` returns 1100 as `assets`.
2. The user transfers 1100 tokens to the protocol.
3. `received` and `maxShares` are both 1100.

The first check `require(maxShares > VIRTUAL_SHARES)` passes because `1100 > 1000`. However, the second check:

```
require(shares <= maxShares - VIRTUAL_SHARES)
```

fails because `shares = 1100` while `maxShares - VIRTUAL_SHARES = 1100 - 1000 = 100`. This causes the transaction to revert, making the function unfunctional when `_totalSupply == 0 || beforeBal == 0`.

Root cause: On the first mint, the transferred asset amount should cover both the requested `shares` and the `VIRTUAL_SHARES`. However, `previewMint` only calculates the amount for the requested `shares` and ignores the additional virtual shares.

Recommendation

Update the calculation to include virtual shares:

```
assets = previewMint(shares + VIRTUAL_SHARES);
```

This ensures that the first mint can succeed while properly accounting for the virtual shares.

Title: L-1 Inconsistent Enforcement Of MAX_PRICE_CHANGE_BPS In Price Bound Logic

Description

There is a logical inconsistency in how MAX_PRICE_CHANGE_BPS is applied across the configuration functions.

When configuring a strategy, the contract enforces that pushThreshold must be less than the constant MAX_PRICE_CHANGE_BPS = 50%, and also less than the maxPriceChangeBps value specific to the strategy ID:

```
function configureStrategy(
    bytes32 strategyId,
    uint256 minUpdateInterval,
    uint256 maxStaleness,
    uint256 pushThreshold,
    uint256 minConfidence
) external onlyOwner {
    if (minUpdateInterval < MIN_UPDATE_INTERVAL) revert UpdateTooFrequent();
    if (maxStaleness > MAX_STALENESS) revert ValueTooStale();
    if (pushThreshold > MAX_PRICE_CHANGE_BPS) revert InvalidPriceChangeBounds();
    if (minConfidence < defaultConfidenceThreshold || minConfidence > 100) revert
LowConfidence();

    uint256 maxChange = maxPriceChangeBps[strategyId];
    if (maxChange == 0) {
        maxChange = MAX_PRICE_CHANGE_BPS; // Use default if not set
    }
    if (pushThreshold > maxChange) {
        revert PushThresholdExceedsMaxChange(pushThreshold, maxChange);
    }

    updateConfigs[strategyId] = UpdateConfig({
        minUpdateInterval: minUpdateInterval,
        maxStaleness: maxStaleness,
        pushThreshold: pushThreshold,
        minConfidence: minConfidence
    });

    emit StrategyConfigured(strategyId, minUpdateInterval, maxStaleness,
pushThreshold);
}
```

However, when setting maxPriceChangeBps for a strategy, the function only enforces that it must be less than BASIS_POINTS = 100%:

```
function setPriceChangeBounds(bytes32 strategyId, uint256 maxChangeBps) external
onlyOwner {
    if (maxChangeBps > BASIS_POINTS) revert InvalidPriceChangeBounds();
    UpdateConfig memory config = updateConfigs[strategyId];
    if (config.pushThreshold > 0 && config.pushThreshold > maxChangeBps) {
        revert PushThresholdExceedsMaxChange(config.pushThreshold, maxChangeBps);
    }

    maxPriceChangeBps[strategyId] = maxChangeBps;
    emit PriceChangeBoundsSet(strategyId, maxChangeBps);
}
```

As a result, it is possible to configure a strategy where `maxPriceChangeBps` exceeds the intended upper limit of `MAX_PRICE_CHANGE_BPS (50%)`. For instance, if `maxPriceChangeBps[5] = 70%`, then a value update with a 60% change would be accepted by `_validatePriceBounds`, despite exceeding the hard-coded “maximum” threshold:

```
function _validatePriceBounds(bytes32 strategyId, uint256 changePercent) internal
view {
    uint256 maxChange = maxPriceChangeBps[strategyId];
    if (maxChange == 0) {
        maxChange = MAX_PRICE_CHANGE_BPS; // Use default if not set
    }

    if (changePercent > maxChange) {
        revert PriceChangeExceedsBounds(changePercent, maxChange);
    }
}
```

This inconsistency undermines the assumption that `MAX_PRICE_CHANGE_BPS` represents an absolute upper limit on price changes.

Recommendation

Either:

1. Rename `MAX_PRICE_CHANGE_BPS` to `DEFAULT_PRICE_CHANGE_BPS` to better reflect its intended purpose, **or**
2. Enforce that `maxPriceChangeBps` must always be ≤ `MAX_PRICE_CHANGE_BPS` to maintain consistent logical bounds.

Title: L-2 Lack Of Upper Bound Check On Nonce In `batchUpdateValues` And `updateValue`

Description

When the functions `batchUpdateValues` or `updateValue` are called, the contract enforces that the provided `nonce` must be strictly greater than the previous nonce of the corresponding strategy. After validation, this new `nonce` is stored as the latest one for that strategy:

```
if (nonce <= lastReport.nonce) revert StaleNonce();

latestReports[strategyId] = ValueReport({
    value: values[i],
    timestamp: block.timestamp,
    confidence: confidences[i],
    nonce: nonce,
    isPush: true,
    lastUpdater: msg.sender
});
```

Moreover, during an emergency update, the function simply increments the existing nonce by one:

```
latestReports[strategyId] = ValueReport({
    value: value,
    timestamp: block.timestamp,
    confidence: 100,
    nonce: latestReports[strategyId].nonce + 1,
    isPush: false,
    lastUpdater: msg.sender
});
```

The issue arises because there is **no upper limit enforced** on the provided `nonce` in either `batchUpdateValues` or `updateValue`. If a transaction provides a nonce equal to `2**256 - 1`, it will be accepted, but subsequent calls to both `emergencyUpdate` and regular update functions will **fail permanently** since the nonce can no longer increment.

In the `OffchainValuationKeeper.py` script, `updateValue` calculates the new nonce by reading the current on-chain value and incrementing it by one, which mitigates this risk for single updates. However, there is no implementation shown for `batchUpdateValues`, leaving uncertainty about how its nonce is derived. Since the function updates multiple strategies simultaneously, it should ideally query the nonce for each strategy and use the highest one plus one as the batch nonce.

Recommendation

Implement a validation check to ensure the provided nonce does not exceed the previous nonce by more than a reasonable bound (e.g., `nonce <= lastReport.nonce + MAX_NONCE_GAP`). This prevents potential lockout scenarios and ensures consistent nonce progression across strategies.

Title: L-3 Missing Validation Between `minUpdateInterval` And `maxStaleness`

Description

When configuring a strategy, the contract enforces that `minUpdateInterval` must be greater than or equal to `MIN_UPDATE_INTERVAL`:

```
if (minUpdateInterval < MIN_UPDATE_INTERVAL) revert UpdateTooFrequent();
```

However, there is **no validation ensuring a logical relationship** between `minUpdateInterval` and `maxStaleness`. This can lead to configuration conflicts — for example, if `minUpdateInterval` is set to **30 hours** while `maxStaleness` is set to **24 hours**, the strategy's value would become stale after 24 hours, yet cannot be updated because the minimum update interval (30 hours) has not elapsed.

Such misconfiguration effectively prevents timely updates and may render the strategy temporarily unusable.

Recommendation

Add a validation to ensure logical consistency by enforcing:

```
require(minUpdateInterval < maxStaleness, "Invalid config: minUpdateInterval must
be less than maxStaleness");
```

This guarantees that strategies can always be updated before their values become stale.

Title: L-4 Inaccurate Idle Asset Calculation In `getIdleAssets`

Description

During allocation, assets are transferred to the adapter. If `executeNow` is set to `false`, the assets are recorded as allocated to a specific strategy ID but remain held within the adapter contract:

```
function allocate(
    bytes memory data,
    uint256 assets,
    bytes4,
    address
) external override onlyVault notPaused returns (bytes32[] memory ids, int256
change) {
    //...

    // Update allocation tracking with transferred amount
    allocations[strategyId] += assets;
    totalAllocations += assets;

    // Add to active strategies if not already present (O(1) operation)
    activeStrategies.add(strategyId);
    // Optionally execute strategy immediately after allocation
    if (executeNow && calls.length > 0) {
        _executeMulticall(strategyId, calls);
    }

    // Return results
    ids = new bytes32 ;
    ids[0] = strategyId;
    change = int256(assets);

    emit AllocationUpdated(strategyId, allocations[strategyId], change);
}
```

The function `getIdleAssets` is designed to return the amount of idle assets not allocated to any strategy, as indicated by its documentation. However, its current implementation simply returns the adapter's total token balance, without excluding tokens that have already been allocated but not yet executed:

```
/// @notice Get idle assets that are not allocated to any strategy
/// @dev L-13 FIX: Provides visibility into unused assets to ensure full
utilization
/// @return idleAssets Amount of assets sitting idle in the adapter
function getIdleAssets() external view returns (uint256 idleAssets) {
    return IERC20(asset).balanceOf(address(this));
}
```

As a result, the returned value may not accurately represent truly "idle" assets. This can mislead off-chain monitoring tools or automation scripts that rely on this data for allocation and rebalancing decisions.

Recommendation

Modify `getIdleAssets` to subtract the total amount of tokens allocated to strategies (but not yet executed) from the adapter's balance. This will ensure the function accurately reflects the amount of genuinely idle assets available for new allocations.

Title: I-1 Lack Of Differentiation Or Logging Between Value Validity Scenarios In `getTotalValue`

Description

When calculating the total value using `getTotalValue`, the function handles four possible scenarios based on data staleness and confidence levels:

```
if (stalenessAge <= maxStaleness && report.confidence >= minConfidence) {
    // Use fresh, high-confidence value
    totalValue += report.value;
} else if (stalenessAge <= ABSOLUTE_MAX_STALENESS && report.confidence >=
minConfidence) {
    // Moderately stale but within absolute limit - use with caution
    totalValue += report.value;
} else if (fallbackValues[strategyId] > 0) {
    // Use fallback value if main value is too stale or low confidence
    totalValue += fallbackValues[strategyId];
} else if (report.value > 0 && stalenessAge <= ABSOLUTE_MAX_STALENESS) {
    // Defense in depth: use last known value within absolute staleness limit
    totalValue += report.value;
}
```

However, three out of the four branches result in the exact same operation — `totalValue += report.value;` — without providing any indication of which path was taken. As a result, callers and off-chain systems receive the same `totalValue` output regardless of whether the data is **fresh**, **moderately stale**, or **near-expired**, with no on-chain signal about the reliability or freshness of the underlying values. The only distinction exists as inline comments, which are not reflected in runtime behavior.

Recommendation

Emit a distinct event or status flag for each scenario (e.g., `ValueUsedFresh`, `ValueUsedStale`, `ValueUsedFallback`) to allow off-chain monitoring systems to accurately assess data quality and confidence in the returned total value.

Title: I-2 Inconsistent Logic Between Confidence Threshold And Update Interval

Description

There is an inconsistency in the contract's logic regarding confidence checks and update timing.

First, when a value is updated, the contract enforces that its confidence must be greater than or equal to the strategy's configured `config.minConfidence`. During emergency updates, the confidence is always set to 100%. Therefore, the only scenario where a value can have a confidence lower than `config.minConfidence` is if the strategy is later reconfigured and the `minConfidence` requirement is increased — making the existing report no longer compliant.

Second, in the `needsUpdate` and `requestUpdate` functions, if a value's confidence falls below `config.minConfidence`, the function either returns `true` or emits an `UpdateRequested` event, signaling that an update is needed. However, an update cannot necessarily be executed immediately because the `config.minUpdateInterval` restriction still applies. This creates a logical conflict where an update is required but cannot be performed due to timing constraints.

Recommendation

Allow updates to bypass the `minUpdateInterval` restriction when the value's confidence is below `config.minConfidence`. This ensures that low-confidence or invalid data can be refreshed promptly, improving data accuracy and system responsiveness.

Title: I-3 Redundant Balance Retrieval In `deallocate` Function

Description

In the `deallocate` function, the adapter's token balance is already fetched and stored in the `adapterBalance` variable at the beginning. However, within the `else` block (executed when `assets > adapterBalance`), the contract redundantly calls `IERC20(asset).balanceOf(address(this))` again to retrieve the same balance:

```
function deallocate(
    bytes memory data,
    uint256 assets,
    bytes4 caller,
    address
) external override onlyVault notPaused returns (bytes32[] memory ids, int256
change) {
    //....

    uint256 adapterBalance = IERC20(asset).balanceOf(address(this));
    uint256 actualAmount;

    if (caller == FORCE_DEALLOCATE_SELECTOR) {
        //....
    } else {
        if (assets <= adapterBalance) {
            // ....
        } else { // When assets > adapter balance
            //....

            uint256 balance = IERC20(asset).balanceOf(address(this));

            //....
        }
    }

    //....
}
```

Fetching the same balance twice is unnecessary and slightly increases gas consumption without any added benefit.

Recommendation

Remove the redundant line

```
uint256 balance = IERC20(asset).balanceOf(address(this));
```

and use the previously defined `adapterBalance` variable for any balance-related calculations. This will make the code cleaner and more efficient.

Title: I-4 Unused Variables: `dailyLimit` And `dailyUsed`

Description

The parameters `dailyLimit` and `dailyUsed` are initialized when setting a strategy but are never utilized anywhere else in the codebase:

```
function setStrategy(
    bytes32 strategyId,
    address agent,
    bytes calldata preConfiguredData,
    uint256 dailyLimit
) external onlyOwner {
    strategies[strategyId] = StrategyConfig({
        agent: agent,
        preConfiguredData: preConfiguredData,
        dailyLimit: dailyLimit,
        lastResetTime: block.timestamp,
        dailyUsed: 0,
        active: true
    });

    emit StrategySet(strategyId, agent, dailyLimit);
}
```

These variables currently have no functional impact and only add unnecessary storage usage.

Recommendation

Remove the unused `dailyLimit` and `dailyUsed` fields from the `StrategyConfig` struct and related logic to simplify the code and reduce gas costs.