Omni Network

Omni SolverNet

Security Assessment Report

Version: 2.0

March, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Omni Network smart contracts in scope. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Omni Network smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Omni Network smart contracts in scope.

Overview

Omni is a chain abstraction layer, allowing developers to make their application available to users and liquidity across all rollups, while only deploying their application on a single rollup.

SolverNet expands the capabilities of Omni Core by implementing intent-driven execution across multiple chains. It allows developers to offload the execution of their application to a network of solvers, who compete to fulfil user intents in a trustless and efficient manner.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Omni Network repository.

The scope of this time-boxed review was strictly limited to files at commit 3ef37ae.

1. `SolverNetInbox.sol`
2. `SolverNetOutbox.sol`
3. `SolverNetExecutor.sol`
4. `SolverNetMiddleman.sol`
5. `lib/*`
6. `utils/*`
7. `interfaces/*`

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Solidity.

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity components of the review, the testing team also utilised the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`
- Aderyn: `https://github.com/Cyfrin/aderyn`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 13 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- High: 2 issues.
- Medium: 2 issues.
- Low: 3 issues.
- Informational: 5 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Omni Network smart contracts in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
OMNI6-01	Unnecessary Reentrancy Protection In `markFilled()` Function Allows Fund Theft	Critical	Resolved
OMNI6-02	Inadequate Gas Estimation Leads To Fund Theft From Memory Expansion Costs	High	Resolved
OMNI6-03	Potential Fund Theft Due To Omni Network Downtime Or Paused `OmniPortal`	High	Resolved
OMNI6-04	Possible Theft Of Contract Balance Due To Lack Of Access Control	Medium	Resolved
OMNI6-05	Block Order Filling For Tokens That Prevent Nonzero Allowances Modifications	Medium	Resolved
OMNI6-06	Unbounded Arrays May Lead To Gas Exhaustion Or Storage Bloat	Low	Resolved
OMNI6-07	Zero Allowance Approval For Certain Tokens Can Prevent Order Filling	Low	Resolved
OMNI6-08	Griefing Attack Via State Manipulation On The Destination Chain Before The Order Being Filled	Low	Closed
OMNI6-09	Possible Zero Value For Immutable Address	Informational	Closed
OMNI6-10	Enhancing Flexibility In Order Execution With Gas Limit Field In `struct Call`	Informational	Closed
OMNI6-11	Preventing Approval Overwriting In Token Expenses During Order Openning	Informational	Closed
OMNI6-12	Adding Margin To Order Filling Deadline	Informational	Closed
OMNI6-13	Miscellaneous General Comments	Informational	Resolved

OMNI6-01	Unnecessary Reentrancy Protection In `markFilled()` Function Allows Fund Theft		
Asset	`SolverNetInbox.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The function `markFilled()` is used to mark an order as filled on the source chain after it has been executed on the destination chain. Although this function uses the `nonReentrant` modifier to protect against reentrancy attacks, its use in this function could allow a malicious user to steal funds.

```
function markFilled(bytes32 id, bytes32 fillHash, address creditedTo) external xrecv nonReentrant
```

Here's how the attack can unfold:

1. **The Setup**: The attacker opens an order, which we will call `orderA`, and it is executed on the destination chain, meaning the attacker receives the necessary tokens on the destination chain. Afterwards, the Solver sends a cross-rollup message via Omni Core to confirm that `orderA` has been fulfilled on the source chain.
2. **The Attack Begins**: While the transaction is pending, the attacker notices the transaction in the mempool that is going to call `OmniPortal.xsubmit()`. Before this transaction is executed on the source chain, the attacker triggers an attack in one of the following two ways:

   1a. The attacker calls `open()` with a malicious token that they control (a fake token). When processing the deposit, this malicious token will be used, allowing the attacker to hijack the control flow during deposit processing.

   ```
   function _processDeposit(SolverNet.Deposit memory deposit) internal {
       if (deposit.token == address(0)) {
           if (msg.value != deposit.amount) revert InvalidNativeDeposit();
       } else {
           deposit.token.safeTransferFrom(msg.sender, address(this), deposit.amount);
       }
   }
   ```

   1b. Alternatively, the attacker can call `close()` using an old order that was not filled before, and whose `fillDeadline + CLOSE_BUFFER` has passed. This allows the attacker to close the order and hijack the control flow during deposit transfer.

   ```
   function _transferDeposit(bytes32 id, address to) internal {
       SolverNet.Deposit memory deposit = _orderDeposit[id];
       if (deposit.amount > 0) {
           if (deposit.token == address(0)) to.safeTransferETH(deposit.amount);
           else deposit.token.safeTransfer(to, deposit.amount);
       }
   }
   ```

3. **Execution of the Attack**: Now that the attacker has control, they call `OmniPortal.submit()` to execute the transaction that was pending in the mempool, which is supposed to mark `orderA` as filled.
4. **Reentrancy Check Triggers a Revert**: When `OmniPortal.xsubmit()` is executed, it calls the internal function `OmniPortal._exec()`, which in turn calls `SolverNetInbox.markFilled()`. However, because of the `nonReentrant` modifier in `markFilled()`, the transaction fails as the attacker already entered the protocol in steps **1a** or **1b**.

5. **Message Loss**: Due to the design of the `OmniPortal`, the failure does not cause a revert in `OmniPortal.xsubmit()`. This means that even though the transaction to mark `orderA` as filled fails on the source chain, it is assumed to have been executed and cannot be retried.
6. **Completion of the Attack**: After this failure, control is handed back to the `SolverNetInbox`, where the transaction from step 1a or 1b continues.

Now that the transaction has ended, the attacker has already received the tokens for `orderA` on the destination chain, but the transaction that would mark it as filled on the source chain has failed and cannot be retried. In other words, the protocol assumes that `orderA` is not filled on the destination chain yet. After waiting for `fillDeadline + CLOSE_BUFFER`, the attacker can call `close()` to retrieve their deposit from the source chain associated with `orderA`. This allows the attacker to steal funds by receiving the deposit on the source chain during closing the order as well as receiving the tokens on the destination chain during filling the order.

The main issue here is the inappropriate use of the `nonReentrant` modifier without valid justification. The `nonReentrant` check in `markFilled()` causes the transaction to fail under these conditions, allowing the attacker to exploit the system.

Recommendations

A possible solution is to remove the `nonReentrant` check from the `markFilled()` function, ensuring that it does not block valid transactions and preventing this type of attack.

Resolution

The development team has addressed this issue by removing the `nonReentrant` modifier from the `markFilled()` function.

This issue has been resolved in commit ea808d7.

OMNI6-02	Inadequate Gas Estimation Leads To Fund Theft From Memory Expansion Costs		
Asset	`SolverNetOutbox.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The function `_fillGasLimit()` is designed to estimate the gas required to mark an order as filled on the inbox. This gas is needed to cover the cost of executing the `_fillHash()` function on the source chain:

```
function _fillHash(bytes32 orderId) internal view returns (bytes32) {
    SolverNet.Header memory header = _orderHeader[orderId];
    SolverNet.Call[] memory calls = _orderCalls[orderId];
    SolverNet.TokenExpense[] memory expenses = _orderExpenses[orderId];

    SolverNet.FillOriginData memory fillOriginData = SolverNet.FillOriginData({
        srcChainId: uint64(block.chainid),
        destChainId: header.destChainId,
        fillDeadline: header.fillDeadline,
        calls: calls,
        expenses: expenses
    });

    return keccak256(abi.encode(orderId, abi.encode(fillOriginData)));
}
```

In this function, the data from `_orderCalls[orderId]` is loaded from storage into memory. However, the function `_fillGasLimit()` calculates the cost of loading from storage as follows:

```
// 2500 gas for Call array length SLOAD + dynamic cost of reading each call.
uint256 callsGas = 2500;
for (uint256 i; i < fillData.calls.length; ++i) {
    SolverNet.Call memory call = fillData.calls[i];
    unchecked {
        // 5000 gas for the two slots that hold target, selector, and value.
        // 2500 gas per params slot (1 per function argument) used (minimum of 1 slot).
        callsGas += 5000 + (FixedPointMathLib.divUp(call.params.length + 32, 32) * 2500);
    }
}
```

This cost estimate includes the gas for loading the call array length, as well as the gas for loading each call's parameters. However, it fails to account for memory expansion costs. For example, if the `mstore` opcode occupies 256 KB of memory space, the cost for memory expansion could be around 155,648 gas. This additional gas cost is not factored into the estimate, despite the 100k base gas being accounted for.

This creates a vulnerability. If an attacker opens an order with a parameter length that is too large, after the order is filled on the destination chain and the attacker receives the tokens, the Solver will attempt to send a cross-rollup message via Omni Core to confirm the fulfillment. However, due to the underestimated gas in `_fillGasLimit()`, the transaction cannot be marked as filled on the source chain, as it requires more gas than expected. The function `_fillHash()` exceeds the estimated gas limit, preventing the Solver from marking the order as filled, and preventing the Solver from claiming the deposited amount.

After the `CLOSE_BUFFER` period, the attacker can close the order and receive the deposited amount, even though the order was filled on the destination chain, leading to the theft of funds.

It is important to note that this scenario could also occur for order expenses.

Recommendations

A potential solution is to include the cost of memory expansion in the gas estimation within `_fillGasLimit()`, based on the Ethereum yellow paper:

```
Gmemory * (max_memory / 32) + floor(max_memory^2 / 524,288)
```

Alternatively, a simplified approach could be:

```
callsGas += 3 * (call.params.length / 32) + (call.params.length * call.params.length / 524,288)
```

Resolution

The development team has addressed this issue by using the recommended formula to include gas costs for memory expansion. The Solady `FixedPointMathLib` `mulDivUp` and `divUp` functions were also incorporated to add a slight buffer to the computation.

This issue has been resolved in commit ea808d7.

OMNI6-03	Potential Fund Theft Due To Omni Network Downtime Or Paused `OmniPortal`		
Asset	`SolverNetInbox.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

If an order is executed on the destination chain, the Solver sends a cross-rollup message via Omni Core to confirm that the intent has been fulfilled on the inbox of the source chain. However, if the Omni network experiences downtime or if the `OmniPortal` remains paused for more than 6 hours, this confirmation message will not be received on the source chain within the expected timeframe. As a result, the order status remains `Pending` instead of being updated to `Filled`.

```
uint256 internal constant CLOSE_BUFFER = 6 hours;
```

```
_upsertOrder(id, Status.Filled, 0, creditedTo);
```

In this scenario, the order owner can call `close()` on the source chain, even though their order has already been fulfilled on the destination chain, allowing them to reclaim their deposited amount. This creates an opportunity for the order owner to exploit the delay in the Omni network and effectively steal funds.

```
function close(bytes32 id) external whenNotPaused(CLOSE) nonReentrant {
    OrderState memory state = _orderState[id];
    SolverNet.Header memory header = _orderHeader[id];

    if (state.status != Status.Pending) revert OrderNotPending();
    if (header.owner != msg.sender) revert Unauthorized();
    if (header.fillDeadline + CLOSE_BUFFER >= block.timestamp) revert OrderStillValid();

    _upsertOrder(id, Status.Closed, 0, msg.sender);
    _transferDeposit(id, header.owner);

    emit Closed(id);
}
```

Recommendations

Relying on a fixed 6-hour buffer is insufficient to mitigate this issue. A better approach would be to check whether `OmniPortal` is paused before allowing order closure or to integrate an oracle to verify the operational status of the Omni network.

Resolution

The development team has addressed this issue with the following note:

"We are fully aware of risk due to Omni network downtime potentially allowing the user to retrieve their deposits if the fill deadline buffer is exhausted. To combat this, we included our pausable functions so that the solver can come and halt the inbox contract's open and close methods in case of a problem. We will have it looking at Omniportal offsets for routes and if there is a sustained lag it will shut down the inbox. We also have Github actions that we can use to trigger this as well. We find 6 hours to be sufficient for this. However, the recommendation to check OmniPortal's pause status was a good one, so we now perform that check as well. If OmniPortal is globally paused, or xsubmit itself is globally paused, or if xsubmit is paused for the specific route, close will revert."

This issue has been resolved in commit ea808d7.

OMNI6-04	Possible Theft Of Contract Balance Due To Lack Of Access Control		
Asset	`SolverNetMiddleMan.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `executeAndTransfer()` function allows the caller to execute a `call`, then transfer the contract's native token balance as well as the balance of a `token` of their choosing to an address specified by the caller.

```
function executeAndTransfer(address token, address to, address target, bytes calldata data) external payable {
        (bool success,) = target.call{ value: msg.value }(data);
        if (!success) revert CallFailed();

        if (token == address(0)) SafeTransferLib.safeTransferAllETH(to);
        else token.safeTransferAll(to);
}
```

However, this function has no access control. This means that anyone is able to invoke it and wipe the contract's balance in the process. Since the contract has both a `receive()` and `fallback()` function, it is assumed that it is expected to hold value at some point in time.

Recommendations

Implement access control on the function by limiting the addresses that are able to invoke it.

Resolution

The development team has addressed this issue with the following note:

"This finding was likely due to a lack of documentation on the Middleman contract. This is an intermediate contract to be used as an intent destination for cases where a contract only returns a token to the caller, and does not offer a address to or address onBehalfOf style parameter. In these cases, we direct the call to the Middleman and have it perform the call, and then we provide the token information as well so it knows what to forward over to the user. It's just a "zap"-like interface to handle these edge cases. It is designed to send the entire balance of the token specified to the user, and only has receive so it can receive ETH if sent by the call, and fallback so that any potential callbacks into it don't cause reverts. We did recognize that a malicious target app or something that app calls into could steal the funds before they're sent to the user, so we applied a reentrancy guard."

This issue has been resolved in commit ea808d7.

OMNI6-05	Block Order Filling For Tokens That Prevent Nonzero Allowances Modifications		
Asset	`SolverNetOutbox.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

When the function `SolverNetExecutor.approve()` is invoked within the `SolverNetOutbox.withExpenses()` modifier, it is expected to grant a nonzero allowance to the specified spender if `expenses` are requested by the owner of the order. However, certain tokens, such as USDT, require that an existing allowance be explicitly set to zero before approving a new nonzero amount. This requirement is not accounted for in the current implementation.

Below is the `approve()` function from USDT, demonstrating this constraint:

```
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {
    // To change the approve amount you first have to reduce the address'
    // allowance to zero by calling `approve(_spender, 0)` if it is not
    // already 0, to mitigate the race condition described here:
    // https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
}
```

Attack Vector:

1. An attacker identifies that a user intends to open an order involving USDT on the destination chain, meaning that `SolverNetExecutor` will approve a nonzero allowance to the user for USDT.
2. The attacker submits an order that manipulates `SolverNetExecutor` to set a nonzero allowance for the user. This is achieved by crafting `orderData.calls` so that `SolverNetExecutor` executes `USDT.approve(user, 1)`:

```
struct Call {
    address target;  // USDT contract address
    bytes4 selector; // approve(address,uint256) function selector
    uint256 value;   // 0 (ETH transfer value)
    bytes params;    // Encoded parameters: user as spender, 1 USDT as amount
}
```

```
SolverNet.Call[] memory calls = orderData.calls;
if (calls.length == 0) revert InvalidMissingCalls();
for (uint256 i; i < calls.length; ++i) {
    SolverNet.Call memory call = calls[i];
    if (call.target == address(0)) revert InvalidCallTarget();
}
```

3. When the attacker's order is executed on the destination chain, it sets the USDT allowance of `SolverNetExecutor` to `1` for the user:

```
_executor.execute{ value: call.value }(
    call.target, call.value, abi.encodePacked(call.selector, call.params)
);
```

4. Later, when the solver attempts to fill the user's order on the destination chain, the modifier `withExpenses` tries to approve a new nonzero amount from `SolverNetExecutor` for the user. However, since the current allowance is already nonzero, the approval fails and reverts, preventing the user's order from being fulfilled:

```
if (spender != address(0)) _executor.approve(token, spender, amount);
```

```
function approve(address token, address spender, uint256 amount) external onlyOutbox {
    token.safeApprove(spender, amount);
}
```

Recommendations

To mitigate this issue, it is advised to use `SafeTransferLib.safeApproveWithRetry()` instead of `approve()`. This function ensures that if a token does not allow changing allowances from a nonzero value, it first resets the allowance to zero before setting the new value.

Resolution

The development team has addressed this issue by replacing `approve()` with `SafeTransferLib.safeApproveWithRetry()` in `SolverNetExecutor.approve()`.

This issue has been resolved in commit ea808d7.

OMNI6-06	Unbounded Arrays May Lead To Gas Exhaustion Or Storage Bloat		
Asset	`SolverNetInbox.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `open()` function allows a user to place an order which contains `orderData` consisting of two arrays, `calls` and `expenses`. These arrays have no cap on their sizes which could lead to issues as follows:

Block Gas Limit
`open()` invokes both `_validate()` and `_openOder()` to process a new order. Both functions contain two `for` loops that iterate over `calls` and `expenses`. The time complexity of the iterations for each function is `O(n+m)`. This means the execution time grows linearly with the number of elements in the `calls` and `expenses` arrays. However, even though the complexity is linear, if a user submits very large arrays for both `calls` and `expenses`, the total gas consumed by these loops can grow such that it exceeds the block gas limit.

Storage Bloat
The `_openOrder()` function pushes values from the `calls` and `expenses` arrays to two state variables:

```
mapping(bytes32 id => SolverNet.Call[]) internal _orderCalls
mapping(bytes32 id => SolverNet.TokenExpense[]) internal _orderExpenses;
```

If these arrays are very large, continuously pushing them into storage can degrade performance and increase gas costs. Especially, as these arrays are not removed from storage after an order has been closed or marked filled. This could potentially be exploited by an attacker to drive up gas costs for future calls to `open()`.

Recommendations

Place a cap on the size of both the `calls` and `expenses` arrays submitted with each order. Also, it could be useful after an order has been closed or marked filled, to update `_orderCalls` and `_orderExpenses` by removing the order accordingly to reduce storage.

Resolution

The development team has addressed this issue by placing a cap on the size of the `calls` and `expenses` arrays to a max of 32 items. Additionally, the team has implemented a `_purgeState()` function to remove order states that are no longer needed.

This issue has been resolved in commit ea808d7.

OMNI6-07	Zero Allowance Approval For Certain Tokens Can Prevent Order Filling		
Asset	`SolverNetOutbox.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

After an order is filled on the destination chain, if the balance of `SolverNetExecutor` and the spender's address are non-zero, the allowance of `SolverNetExecutor` to the spender is set to zero:

```
if (tokenBalance > 0) {
    address spender = expense.spender;
    if (spender != address(0)) _executor.approve(token, spender, 0);
    _executor.transfer(token, msg.sender, tokenBalance);
}
```

This logic presents an attack vector for certain tokens that could prevent a user's order from being filled.

For example, consider the token BNB. The BNB implementation on Ethereum does not allow approval of zero allowance:

```
function approve(address _spender, uint256 _value)
    returns (bool success) {
    if (_value <= 0) throw;
    allowance[msg.sender][_spender] = _value;
    return true;
}
```

(The token contract code can be viewed here).

In this case, before the user's order is filled on the destination chain (e.g., Ethereum), an attacker could simply transfer 1 wei of BNB to the `SolverNetExecutor`. When the user's order is filled, the BNB balance of `SolverNetExecutor` will be non-zero, triggering the `if` condition. If the spender address is nonzero (as defined by the user), the system will attempt to approve a zero allowance. However, due to the BNB token's implementation, this transaction will revert.

Recommendations

A solution is to this issue could be to use a try/catch mechanism to handle such cases.

Resolution

The development team has addressed this issue by adding a `try/catch` mechanism to handle the case where the token does not allow zero allowance approval.

This issue has been resolved in commit ea808d7.

OMNI6-08	Griefing Attack Via State Manipulation On The Destination Chain Before The Order Being Filled		
Asset	`SolverNetExecutor.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

When a user opens an order on the source chain, the Solver simulates the order internally to verify its validity. This creates a potential griefing/DoS attack vector.

An attacker can exploit this by submitting a high gas-consuming order. The Solver, during its internal simulation, ensures the order is valid by executing the transaction and confirming that the target contract is successfully called. The Solver then attempts to fulfill the order on the destination chain. However, before `SolverNetOutbox.fill()` is executed, the attacker can submit another transaction on the destination chain that alters the state of the target contract, causing it to revert upon execution.

As a result, the Solver's transaction fails, preventing a cross-rollup message from being sent via Omni Core to confirm the fulfillment of the intent.

```
function execute(address target, uint256 value, bytes calldata data) external payable onlyOutbox {
    (bool success,) = payable(target).call{ value: value }(data);
    if (!success) revert CallFailed();
}
```

This means the Solver incurs gas costs on the destination chain but does not receive any payment on the source chain. Once the `CLOSE_BUFFER` time expires, the attacker can call `close()` to reclaim their deposited funds.

Recommendations

A solution is to disregard the outcome of the target contract call, as long as the Solver provides sufficient value, token amount, and gas.

Resolution

This issue has been acknowledged by the development team with the following note:

"We decided to ignore this finding, as initially we will be the only permissioned solver. Additionally, we do transaction simulation in the solver to determine if the transaction executes as expected. If execution isn't possible, we reject so the user can get their funds back. We do not waste gas on fills. For now, we will disregard this finding due to these considerations. "

OMNI6-09	Possible Zero Value For Immutable Address
Asset	`SolverNetExecutor.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The constructor is used to initialise `outbox`. However, this variable is `immutable` and there is no check to ensure that the constructor's `_outbox` parameter which is used to initialise is non-zero before assigning its value.

Recommendations

Consider adding a check to ensure that the `_outbox` address parameter is non-zero before assigning its value to `outbox`.

Resolution

This issue has been acknowledged by the development team with the following note:

"This finding cannot happen as the Executor contract is only deployed via the Outbox. Because of this, this address should always be populated. If we opt to deploy the Executor separately, we will address this finding by implementing the check".

OMNI6-10	Enhancing Flexibility In Order Execution With Gas Limit Field In `struct Call`
Asset	`SolverNet.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

It might be beneficial to include a gas limit field in the `struct Call`, as it would provide users with more flexibility when specifying the call during order fulfillment on the destination chain. Currently, the `Call` struct is defined as follows:

```
struct Call {
    address target;
    bytes4 selector;
    uint256 value;
    bytes params;
}
```

During execution, the call is made as shown below:

```
_executor.execute{ value: call.value }(
    call.target, call.value, abi.encodePacked(call.selector, call.params)
);
```

Recommendations

To improve flexibility, it is recommended to add a `gasLimit` field to the `Call` struct, as follows:

```
struct Call {
    address target;
    bytes4 selector;
    uint256 value;
    bytes params;
    uint256 gasLimit;
}
```

Then, the execution would need to be adjusted accordingly:

```
_executor.execute{ value: call.value, gas: call.gasLimit }(
    call.target, call.value, abi.encodePacked(call.selector, call.params)
);
```

Resolution

This issue has been acknowledged by the development team with the following note:

"We are factoring in gas costs within the intent quote. We do not want to complicate integration efforts or UX by requiring gas limits be specified, and have opted for simply quoting properly for the action via our API. If a user manually submits a quote that our solver deems insufficient, it will reject it. Because of this design, this suggestion was not implemented."

OMNI6-11	Preventing Approval Overwriting In Token Expenses During Order Openning
Asset	`SolverNetInbox.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

When opening an order, the user can set the expenses, which include the token address, the spender address, and the amount. The following validation is performed to ensure the expenses are valid:

```
// Validate SolverNet.OrderData.Expenses
SolverNet.TokenExpense[] memory expenses = orderData.expenses;
for (uint256 i; i < expenses.length; ++i) {
    if (expenses[i].token == address(0)) revert InvalidExpenseToken();
    if (expenses[i].amount == 0) revert InvalidExpenseAmount();
}
```

However, if a user requests an order with two expenses involving the **same token** and **same spender**, the approvals from `SolverNetExecutor` to the spender on the destination chain will overwrite each other, leading to an unexpected result.

For example, if the user specifies the following expenses:

```
orderData.expenses = [TokenExpense(spender: Bob, token: tokenA, amount: 5), TokenExpense(spender: Bob, token: tokenA, amount: 7)]
```

On the destination chain, the order filling process will first approve 5 `tokenA` to Bob, then approve 7 `tokenA` to Bob, resulting in a total approval of 7 `tokenA` to Bob. However, the user's intended goal might have been to approve 12 `tokenA` to Bob.

```
token.safeTransferFrom(msg.sender, address(_executor), amount);
// We remotely set token approvals on executor so we don't need to reprocess Call expenses there.
if (spender != address(0)) _executor.approve(token, spender, amount);
```

Recommendations

To prevent this issue, you can either check for unique token and spender pairs on the source chain or use `increaseAllowance` on the token, if possible.

Resolution

This issue has been acknowledged by the development team with the following note:

"This is a good finding, but because we expect the majority of intents to be filed via our API/SDK, this case will be prevented by our offchain tooling. Our documentation will address this edge case, but enumerating all expenses onchain is seen as an unnecessary cost."

OMNI6-12	Adding Margin To Order Filling Deadline
Asset	`SolverNetInbox.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

When an order is opened, a filling deadline is set:

```
function _validate(OnchainCrossChainOrder calldata order) internal view returns (SolverNet.Order memory) {
    // Validate OnchainCrossChainOrder
    if (order.fillDeadline <= block.timestamp) revert InvalidFillDeadline();
```

It is logical for the Solver to wait for finality on the source chain before filling the order on the destination chain. This reduces the risk of reorgs and reversals, ensuring that transactions are securely settled. Therefore, if an order is opened with a `fillDeadline` earlier than `block.timestamp + finality`, the order will expire before it can be filled on the destination chain (assuming the Solver waits for finality before proceeding). For example, if the `fillDeadline` is `block.timestamp + 1`, the order cannot be filled on the destination chain. It will either be rejected by the Solver or, after the `CLOSE_BUFFER` time, the owner can close the order.

Recommendations

To address this, it is recommended to add a margin to the filling deadline when opening an order:

```
if (order.fillDeadline <= block.timestamp + 120) revert InvalidFillDeadline();
```

Resolution

This issue has been acknowledged by the development team with the following note:

"We are not adding a margin to fill deadlines because we are leaving this up to the solvers. We not only want users to be able to say "only do this if you can fill it in 30 seconds", it should be up to the solver to decide if they want to take on the finality risk."

OMNI6-13	Miscellaneous General Comments
Asset	All contracts
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Unnecessary Use Of `Initializable` In `SolverNetMiddleman.sol`**
   Related Asset(s): SolverNetMiddleman.sol
   This contract is marked as `Initializable` but does not contain an `initialize()` function for initialization.
   Consider adding an `initialize()` function to setup an owner or other roles for the contract. Otherwise, remove `Initializable` from the contract's definition.
2. **Possible Array Index Out Of Bounds During Loop Iteration**
   Related Asset(s): SolverNetInbox.sol
   The `setOutboxes()` function iterates over two arrays, `chainIds` and `outboxes`, to update the `_outboxes` state variable with the outbox address for each destination chain. During this process, it reads the current index from both arrays. However, if both arrays are not the same length then an index out of bounds error will occur and the call will revert.
   Consider adding a check to ensure that both `chainIds` and `outboxes` are the same length.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

These issues were addressed by the development team by removing `Initializable` from the `SolverNetMiddleman.sol` contract and adding a check to ensure that the `chainIds` and `outboxes` arrays are the same length in the `setOutboxes()` function.

This issue has been resolved in commit ea808d7.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

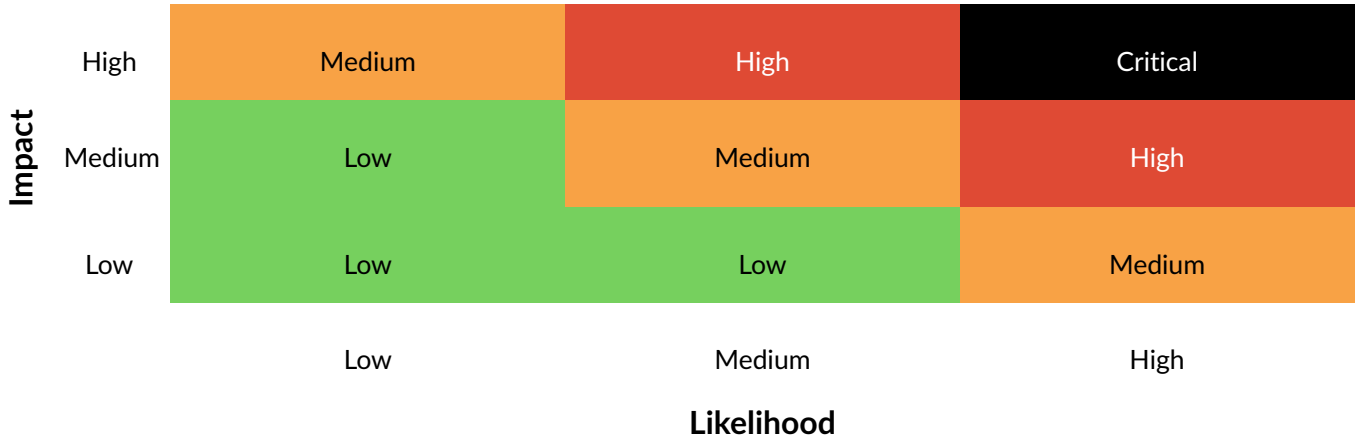

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].