PUBLIC

Code Assessment

# of the OmniBridge Smart Contracts

April 27, 2021

Produced for

POA

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Sir or Madam,

First and foremost we would like to thank POA Network for giving us the opportunity to assess the current state of their OmniBridge system. This document outlines the findings, limitations, and methodology of our assessment.

We hope that this assessment provides more insight into the current implementation and provides valuable findings. We are happy to receive questions and feedback to improve our service and are highly committed to further support your project.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	8
• Code Corrected	6
• Risk Accepted	2
Low-Severity Findings	8
• Code Corrected	4
• Specification Changed	1
• Risk Accepted	2
• Acknowledged	1

2 Assessment Overview

In this section we briefly describe the overall structure and scope of the engagement including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the OmniBridge repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	March 1 2021	ea0ffa6f015da024306d80f61bd271c4268b1f7a	Initial Version
2	April 8 2021	9e602a3719e32feabc18fc387b9474acfa28cfe2	Version with fixes

For the solidity smart contracts, the compiler version `0.7.5` was chosen.

2.1.1 *Excluded from scope*

The Arbitrary Message Bridge (AMB) smart contracts were not part of the scope.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section we have added a version icon to each of the findings to increase the readability of the report.

OmniBridge is a system of smart contracts that allows cross-chain token transfers between Ethereum-compatible blockchains.

Cross-chain communication is based on the Arbitrary Message Bridge (AMB), which replays messages between two blockchains: the **home blockchain** and the **foreign blockchain**. Messages can originate at either blockchain, and are initially stored in the **AMB mediator contract** on that blockchain. Each message includes a target address (on the target blockchain) and may include extra data parameters. Then, dedicated oracles relay the messages to the AMB mediator contract on the other blockchain. A relayed message is authenticated if enough oracles confirm its authenticity.

OmniBridge uses this message-relay system to establish token transfers between the home and the foreign blockchains. The transfers are implemented by pairing a token contract, **the native token**, on one of the blockchains, with a token contract, **the bridged token**, on the other blockchain. The native token is independent of the OmniBridge system, while the bridged token is automatically deployed by the system. Transfers are mediated by two AMB mediator contracts on every side of the bridge.

There are two message flows:

- For native -> bridged transfers:
    1. the transfer sender approves native token transfer to the mediator contract;
    2. the transfer sender invokes the mediator contract: 1. the mediator contract transfers to itself the specified amount of tokens; 2. the tokens are now locked in the mediator; 3. the mediator sends a transfer message through the AMB;

3. when the message is received, the mediator on the other side mints the specified tokens of the bridged token to the receiver of the transfer.

- For bridged -> native transfers, the flow is reversed: tokens are burned on the sender (bridged) side and unlocked on the receiver (native) side.

As mentioned, the bridged token counterpart of a native token is deployed on-demand. The native token starts in a **deploy unacknowledged state**. In this state, a cross-chain transfer generates an additional request to deploy the bridged token counterpart, which until this moment is non-existent. After the deployment request is received, the bridged counterpart is deployed by the `TokenFactory` contract. This leads to the native token state being updated to **deploy acknowledged**.

In addition to deployment, both tokens need to have their operational limits set. For native tokens this happens upon the first cross-chain transfer, and for their bridged counterparts---after deployment. In particular, the minimum transfer amount per transaction is set, which promotes the token from **unregistered** to **registered** status. The registered status is required for the completion of cross-chain transfers.

To reduce the cost of deployment, all bridged tokens share the same ERC677 implementation. Each bridged token is actually a proxy that stores the token balances, but delegates all calls to a predeployed ERC677 contract. Thus, all proxies have very small code and deployment is relatively cheap.

The behavior of the mediator contracts on the home blockchain and on foreign blockchain are mostly identical. However, differences between the home and the foreign blockchains can make it economically more efficient to perform certain operations on the home blockchains. (For example, because of lower transaction fees, faster throughput, lower latency, etc.) This results in the following differences between the home OmniBridge and the foreign OmniBridge:

- Fees are collected and payed in the home contract only.
- Gas limit on the foreign contract is stored on the contract itself and can be modified by owner.
- Gas limit on the home contract is queried from a separate SelectorTokenGasLimitManager contract. This contract can return an exact limit for token and function based on the message data. Tokens not configured on the SelectorTokenGasLimitManager will use default gas limits.

The same considerations also make the AMB transaction handling different. On the home side, oracles will submit the incoming transaction. On the foreign side, oracles just provide the necessary signatures before it is actually submitted to the foreign blockchain by the users (senders/receivers) themselves. This way the higher foreign gas fees will be payed by the users.

2.2.1 Trust Model

Users of the OmniBridge need to trust in different parties:

- xDai Validators: They are trusted to avoid any forks of more than 12 blocks and generally follow the protocol honestly.
- OmniBridge Contract Administrators: These administrator can change numerous values and thereby execute a lot of control as described in some findings below.
- TokenFactory Contract Administrators: The administrators controlling the TokenFactory can determine the base image that is used for all bridged tokens. By choosing a malicious image, they can steal the bridged funds.
- FeeManager Contract Administrators: The administrators of the FeeManager contract can set the fees and also exempt addresses from fee payments. By setting extremely high fees, they can essentially steal the funds that are sent across the bridge.
- AMB Validators: The AMB Validators are fully trusted to correctly relay information from one side to the other.
- Ethereum miners: They are trusted to avoid any forks of more than 12 blocks and generally follow the protocol honestly.

Generally, honest majorities are required from all of these parties.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. We draw attention to the fact that due to inherent limitations in any software development process and software product an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third party infrastructure which adds further inherent risks as we rely on the correct execution of the included third party technology stack itself. Report readers should also take into account the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severities. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved, have been moved to the Resolved Findings section. All of the findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	2

- Administrators Can Make Non-Native Tokens Native and Native Tokens Non-Native Risk Accepted
- Tokens With More Than One Token Address Can Be Stolen by Admins Risk Accepted

Low-Severity Findings	3

- Documentation Mismatches Acknowledged
- Function onTokenTransfer Reentrancy Case Risk Accepted
- Incompatible Tokens Risk Accepted

5.1 Administrators Can Make Non-Native Tokens Native and Native Tokens Non-Native

Correctness Medium Version 1 Risk Accepted

When the function `setCustomTokenAddressPair` is called, the following checks are being performed:

```
require(!isTokenRegistered(_bridgedToken));
require(nativeTokenAddress(_bridgedToken) == address(0));
require(bridgedTokenAddress(_nativeToken) == address(0));
```

However, there is no check that the `_nativeToken` is not bridged token, i.e.:

```
require(nativeTokenAddress(_nativeToken) == address(0));
```

This can create a weird condition where the bridged token is again a native token. Once this occurs, the bridge fails to function correctly, as the bridged tokens are now handled as native tokens.

Similarly, administrators can also register an existing native token, as a non-native token. Consider the following example:

1. A native token T exists, which has already been bridged and where tokens of type T are locked up inside the mediator contract.
2. An administrator call `setCustomTokenAddressPair` with T as the `_bridgedToken` and some other fake token F as the supposedly native token on the other side.

3. The attacker transfers a lot of F token (which can be freely minted) over the brige and thereby unlocks the T tokens.

This allows administrators to steal all native tokens held by the bridge.

However, the overall risk is rather low as only administrators can call `setCustomTokenAddressPair`.

Risk accepted:

To address this problem, POA Network added a comment saying that the function arguments should be manually validated by the administrator, as no easy solution is available.

5.2 Tokens With More Than One Token Address Can Be Stolen by Admins

Security Medium Version 1 Risk Accepted

Tokens that have more than one address, through which they can be called, can be stolen when they are bridged. An example for such a token is TUSD. The attack would work as follows:

1. The token is already bridged using the first token address. An amount X has been transferred across the bridge.
2. An attacker bridges the token using another token address. The attacker also bridges X tokens. Now the mediator balance on the native side is X for both token addresses. However, the actual balance, when queried from `balanceOf` is 2*X for both of them.
3. The attacker colludes with the administrators, which trigger a call to `fixMediatorBalance` on the native side and withdraw X amount of tokens.
4. Then, the attacker can withdraw X tokens, by sending back the bridged tokens.

Overall, turned X tokens into 2*X tokens, when ignoring fees. The attacker managed to withdraw the full amount of bridged tokens. At the time of writing 118,000 TUSD have been bridged which are at risk under such an attack.

Risk accepted:

No code changes were done. POA Network added a warning comment to `fixMediatorBalance` method.

5.3 Documentation Mismatches

Correctness Low Version 1 Acknowledged

The following mismatches with the documentation or within the documentation were found:

1. The definition of native is different in the code and in the documentation: https://docs.tokenbridge.net/about-tokenbridge/features#chain-and-network-definitions

   In the code, native refers to the origin of the token contract, in the documentation to the home side of the network.
2. Some documentation items mention a `requiredBlockConfirmations` of 8 while others mention 12.

Acknowledged:

Documentation will be re-worked with help of a technical writer.

5.4 Function `onTokenTransfer` Reentrancy Case

Security Low Version 1 Risk Accepted

The main contracts have a `lock()` function with a corresponding variable that aims as a reentrancy guard. In case when `lock()` is true, the `onTokenTransfer` function will silently accept the funds. This can lead to a reentrancy that can break some invariants of the contract. In case, the callback happens during the `safeTransferFrom` in the `_relayTokens` function, the `from` address can perform a token transfer to the Bridge contract. Note that the same or a different token can be used. Such callbacks during `safeTransferFrom` can occur with tokens that implement the ERC777 or similar standards. Because the received tokens are silently accepted and `_setMediatorBalance` is not called, the `mediatorBalance` won't track the balance correctly.

Risk accepted:

The described behaviour is acceptable, as ERC-777 tokens are not supported by the OmniBridge.

5.5 Incompatible Tokens

Design Low Version 1 Risk Accepted

The following token types are incompatible with Omnibridge:

- Rebasing tokens: If the balance of a token can change while it is stored inside the mediator contract, then basic assumptions no longer hold. Hence, such tokens as Ampleforth should not be bridged as the bridging might not be reversible.
- Special transfer fees: This report already contains issues regarding "regular transfer fees", where upon transfer of X tokens, X-F tokens are transferred, while F tokens are paid to the fee receiver. In case of transfer fees, where upon transfer of X tokens, X+F tokens are subtracted from the senders balance and X tokens arrive at the receiver, the Omnibridge contracts will fail as they do not account for such fees.
- Malicious tokens: Obviously, any malicious token contracts that do not follow sensible guidelines so that for example, balances can be arbitrarily can freely manipulated, cannot be bridged in a meaningful manner.

Users should be warned not to bridge such tokens.

Risk accepted:

POA Network manually reviewed the most important tokens to ensure their compatibility and will monitor the bridge and the bridged tokens. Furthermore, appropriate warnings will be added inside the UI.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	6

- Decimals in bridgeSpecificActionsOnTokenTransfer Are Not Used Code Corrected
- ERC20 Function Calls Ignore Return Values Code Corrected
- No Canonical Definition of Calldata for onTokenTransfer Code Corrected
- Safe Transfers Are Not Used for All Token Transfers Code Corrected
- Transferred Values in Case of Relaying Tokens With Fees Code Corrected
- OmnibridgeFeeManager Fee Distribution Reverts in Case of Tokens With Transfer Fees Code Corrected

Low-Severity Findings	5

- Code Simplification Possible Specification Changed
- Name Collision Among Bridged Tokens With Different Origins Code Corrected
- Reentrancy Into AMB Code Corrected
- Restriction to Static Call Code Corrected
- Superfluous Loads From Storage Code Corrected

6.1 Decimals in `bridgeSpecificActionsOnTokenTransfer` Are Not Used

Design Medium Version 1 Code Corrected

In both Home and Foreign OmniBridge contracts the decimals are queried in the `bridgeSpecificActionsOnTokenTransfer` function during the token relaying. But this data is used only in few cases within this function:

- Token is not registered and limits need to be initialised.
- Token is native to the current side of the bridge and its deployment is not yet acknowledged.

In case of non-native, acknowledged or initialised Tokens the queried decimals won't be used. Because such cases are the most common ones, the unused data introduces extra gas costs that could be avoided.

Code corrected:

The use of `TokenReader.readDecimals()` was refactored as so it is being called only when `deployAndHandleTokens` messages are sent.

6.2 ERC20 Function Calls Ignore Return Values

Design Medium Version 1 Code Corrected

The ERC20 specification states:

```
Callers MUST handle false from returns (bool success). Callers MUST NOT assume that false is never returned!
```

In some calls to the ERC20 tokens those return values are ignored:

- `IBurnableMintableERC677Token(_token).mint(address(manager), fee)` in `_distributeFee` function.
- `IBurnableMintableERC677Token(_token).transfer(address(manager), fee)` in `_distributeFee` function.
- `IBurnableMintableERC677Token(_bridgedToken).mint(address(this), 1)` in `setCustomTokenAddressPair` function.
- `_getMinterFor(_token).mint(_recipient, _value)` in `_releaseTokens` function.

In most cases that happens during the calls to non-native Tokens that were deployed via the factory. But due to the `setCustomTokenAddressPair` function the non-native contracts can have any behavior and the return values need to be checked explicitly.

Code corrected:

All calls to transfer and mint function now check the return values.

6.3 No Canonical Definition of Calldata for `onTokenTransfer`

Correctness Medium Version 1 Code Corrected

The function `onTokenTransfer` uses inline assembly to read the receiver and calldata from the calldata arguments. The assembly strongly relies on some assumptions about the argument encoding of the Solidity. One of them is that there are no "garbage bits" between the byte offset of the `bytes calldata _data` variable and the length field of the `bytes calldata _data` argument. This assumption will hold true in most cases, but is not guaranteed to hold. This assumption can be eliminated letting the compiler copy the `_data` into the memory and dealing with it there. Full expectations about the expected information in the `_data` argument must be properly documented, to avoid the misinterpretation of the interface.

```
function onTokenTransfer(
    address _from,
    uint256 _value,
    bytes calldata _data
) external returns (bool) {
```

Code corrected:

For the relevant onTokenTransfer function, the calldata location of the _data variable was replaced with the memory location. Hence, the ABI parsing is performed by the compiler and only afterwards data is being parsed in inline assembly.

6.4 Safe Transfers Are Not Used for All Token Transfers

Design Medium Version 1 Code Corrected

For some transfers of ERC20 Tokens the SafeERC20 functions are not used. This includes:

- The function _distributeFee in OmnibridgeFeeManagerConnector contract.
- The function distributeFee in OmnibridgeFeeManager contract.

The first case only appears for non-native tokens at the Home side of the bridge, which in most cases should be ERC677 deployed by Factory. But due to the setCustomTokenAddressPair function, there are possible conditions when any other token can be called with this transfer.

Code corrected:

All calls to the transfer function were replaced by a safe wrapper.

6.5 Transferred Values in Case of Relaying Tokens With Fees

Design Medium Version 1 Code Corrected

In a scenario with token relaying, the _relayTokens function is executed. A user provided _value is then transferred to the bridge contract via safeTransferFrom. If the token has fees on transfer (e.g. USDT-not currently charged, PAXG), the actual transferred value will be smaller than the bridged value. This will effectively break the invariant Balance of bridge == total supply of bridged token.

Code corrected:

This has been corrected by measuring the actually transferred token amount.

6.6 OmnibridgeFeeManager Fee Distribution Reverts in Case of Tokens With Transfer Fees

Design Medium Version 1 Code Corrected

As part of the internal function _distributeFee of the OmnibridgeFeeManagerConnector contract calls the token contract to transfer or mint the **fee** amount to the manager. In case the relevant token contract is native to Home side it might have transfer fees. Then, a value less than **fee** will be moved during the transfer to the OmnibridgeFeeManager. Later the OmnibridgeFeeManager tries to distribute this **fee** amount using distributeFee function. Because the actually transferred value will be smaller in case of Tokens with transfer fees, the `OmnibridgeFeeManager` will not have enough assets to perform the reward distribution with the required values.

Hence, the whole transaction will fail and such tokens cannot be moved across the bridge.

Code corrected:

The code has been rewritten so that

1. The `OmnibridgeFeeManager` determines the amount of fees to distribute by calling `token.balanceOf(address(this))`.
2. Failure of the transfer/mint operation during the fee distribution will not fail the `Omnibridge` message processing.

6.7 Code Simplification Possible

Design Low Version 1 Specification Changed

The following code can be simplified:

```
if (_token == address(0xb7D311E2Eb55F2f68a9440da38e7989210b9A05e)) {
    // hardcoded address of the TokenMinter address
    return IBurnableMintableERC677Token(0xb7D311E2Eb55F2f68a9440da38e7989210b9A05e);
}
return IBurnableMintableERC677Token(_token);
```

The `if` clause can be entirely omitted.

Specfication changed:

Before the `OmniBridge` is deployed for the ETH-xDAI instance the contract address in this check can be replaced with the actual minter `0x857DD07866C1e19eb2CDFceF7aE655cE7f9E560d` of the STAKE token on the xDai chain. For other bridges this check is either removed at all or did not have any significant impact. A comment was added into the code to bring more clarity why this check is needed.

6.8 Name Collision Among Bridged Tokens With Different Origins

Design Low Version 1 Code Corrected

When the bridge creates token contracts on the Home chain, the " on xDai" string is appended to the Foreign token name. In case of multiple bridges to different Foreign chains, different tokens that have the same name on different Foreign chains, will have same names on the Home chain. As an example, "1INCH Token" from Ethereum Mainnet and Binance Smart Chain will both have the "1INCH Token on xDai" name on the Home chain. While that has no direct code-related problems, this increases the human error chance during the user interactions.

Code corrected:

Newly deployed Omnibridge contracts are using "from X" names where X is the respective blockchain. The Blockscout interface also renames such tokens in the UI. Unfortunately, it is not possible to change token names for already existing tokens. However such collisions are being mitigated in the UI.

6.9 Reentrancy Into AMB

Security Low Version 1 Code Corrected

When the Arbitrary Message Bridge contract receives a message from the other side, the following is performed code is used to execute the message call:

```
setMessageSender(_sender);
setMessageId(_messageId);
setMessageSourceChainId(_sourceChainId);

...

bool status = _contract.call.gas(_gas)(_data);
setMessageSender(address(0));
setMessageId(bytes32(0));
setMessageSourceChainId(0);
```

The called contracts can query the information such as `messageId` and `messageSender`. These information provide important authorization for the called contracts. As there is no reentrancy guard on this function, this code can be reentered in the following way:

1. Call A is made, correct information for A is available
2. A triggers the reentrancy and call B is made, now B is executing and the correct information for B is available
3. The call B completes and the information are reset to 0
4. The execution of A continues, but now the queried information will be 0

Hence, it is possible that during the execution of a passed message the wrong context, namely 0 is returned when queried from the AMB contract. Furthermore, events are emitted in an interlaced order which might confuse connected systems.

Please note the AMB contracts were outside of the scope of this review, however, we still note this as it can affect the OmniBridge.

Code corrected:

The issue was fixed in https://github.com/poanetwork/tokenbridge-contracts/pull/577. It ensures that no other message relay is currently being processed.

6.10 Restriction to Static Call

Security Low Version 1 Code Corrected

The contract contains the following code to determine the `upgradabilityOwner`:

```
address(this).call(abi.encodeWithSelector(UPGRADEABILITY_OWNER))
```

However, this function is defined as a `view` function:

```
function upgradeabilityOwner() external view returns (address);
```

Hence, a `staticcall` can be used to avoid unexpected state modifications.

Code corrected:

The `call` was replaced with a `staticcall`.

6.11 Superfluous Loads From Storage

Design Low Version 1 Code Corrected

The Omnibridge contracts sometimes contain code like this:

```
require(!bridgeContract().messageCallStatus(_messageId));
require(bridgeContract().failedMessageReceiver(_messageId) == address(this));
require(bridgeContract().failedMessageSender(_messageId) == mediatorContractOnOtherSide());
```

As there is a storage load (SLOAD) inside the `bridgeContract()` function, this SLOAD will be executed three times in this case. Due to the about-to-be introduced EIP-2929 the additional costs of extra SLOADs from the same location are significantly lowered, but it could still be avoided to do it.

Code corrected:

The return value of `bridgeContract()` was saved in a local variable to avoid repeated calls.

7 Notes

We leverage this section to highlight potential pitfalls which are fairly common when working Distributed Ledger Technologies. As such technologies are still rather novel not all developers might yet be aware of these pitfalls. Hence, the mentioned topics serve to clarify or support the report, but do not require a modification inside the project. Instead, they should raise awareness in order to improve the overall understanding for users and developers.

7.1 Differing Token Values

Note **Version 1**

The OmniBridge has limits on transfers per token. This means that only a certain amount of tokens can be transferred per transaction and per day. Generally, this limits are initialized as a number of tokens. Obviously, a certain number of tokens of one type can have a very different value than the same number of tokens from another type. Hence, these limits need to be carefully monitored.

7.2 Function `requestFailedMessageFix` Performs Multiple Calls to `bridgeContract`

Note **Version 1**

When a user detects a failed, bridged message, the function `requestFailedMessageFix` can be used to fix the failed call. Therefore, three pieces of information are needed which are currently loaded like this:

```
    require(!bridgeContract().messageCallStatus(_messageId));
    require(bridgeContract().failedMessageReceiver(_messageId) == address(this));
    require(bridgeContract().failedMessageSender(_messageId) == mediatorContractOnOtherSide());

This code is execute both on Home and Foreign bridges.
```

Note that there are two levels of inefficiency here. First of all three separate calls are made, even though these information are generally always queried together. Second, this information is spread amount three storage slots, and hence requires three costly SLOADs, even though two storage slots would easily suffice, as only 321 bit of data are stored.

However, as this needs to be resolved within the AMB contracts, it is outside the scope of this code review.

7.3 Limits Can Be Compressed in Storage

Note **Version 1**

There are three storage slots being consumed on both sides of the bridge for the following information:

```
uintStorage[keccak256(abi.encodePacked("dailyLimit", _token))] = _limits[0];
uintStorage[keccak256(abi.encodePacked("maxPerTx", _token))] = _limits[1];
uintStorage[keccak256(abi.encodePacked("minPerTx", _token))] = _limits[2];
```

These information are often accessed together. Given the value ranges they could probably be compressed into two storage slots. This would also provide gas savings on the foreign side as it would avoid a costly SLOAD.

7.4 Proxy Fallback Redundant Operations

Note Version 1

The Proxy contract does some redundant operations, such as:

- `let ptr := mload(0x40)`
- `mstore(0x40, add(ptr, returndatasize()))`

Preserving the free memory slot pointer at 0x40 is important when the assembly code is used together with Solidity code. But in case of the Proxy contract, this can be skipped, as no solidity code is executed after the assembly block.

7.5 Redundant Work Performed as Part of `totalSpentPerDay`

Note Version 1

The function `bridgeSpecificActionsOnTokenTransfer` has the following code, that checks and adjusts the `totalSpentPerDay` limit for a particular token.

```
require(withinLimit(_token, _value));
addTotalSpentPerDay(_token, getCurrentDay(), _value);
```

The code of those 2 functions are quite similar.

```
function withinLimit(address _token, uint256 _amount) public view returns (bool) {
    uint256 nextLimit = totalSpentPerDay(_token, getCurrentDay()).add(_amount);
    return
        dailyLimit(address(0)) > 0 &&
        dailyLimit(_token) >= nextLimit &&
        _amount <= maxPerTx(_token) &&
        _amount >= minPerTx(_token);
}


function addTotalSpentPerDay(
    address _token,
    uint256 _day,
    uint256 _value
) internal {
    uintStorage[keccak256(abi.encodePacked("totalSpentPerDay", _token, _day))] = totalSpentPerDay(_token, _day).add(
        _value
    );
}
```

The function `withinLimit`, that is executed first, reads, increases and checks limits. The function `addTotalSpentPerDay` reads, increases and writes the increased value for the limit. This is a small redundancy that can potentially be eliminated.

7.6 Reentrancy Lock Is Gas Inefficient

Note Version 1

The main contracts have a reentrancy guard. Setting and releasing this guard inside OmniBridge contracts is done via storage of a boolean **true/false**.

Please note that using locks which switch between the values 0 and 1 is more expensive than switching between the values 1 and 2 in case of a reverting transaction. However, the correct choice of this values in the future will also be affected by the currently discussed EIP-3298 which is concerned about the removals of refunds.

Based on EIP-2929 it would also be beneficial if the reentracy lock value would be packed into the same storage slot with another variable, but that is hard due to the chosen storage layout.

7.7 State of Implementation Contract

Note Version 1

With proxied contracts, the state generally resides in the proxy while the code resides inside the implementation contract. In principle, the state of the implementation contract is meaningless, unless the code contains selfdestruct, callcode or delegatecall opcodes. Neither of these opcodes can be found inside the current Omnibridge contracts. However, we would still recommend to make the initialization of the state of the implementation contract part of the deployment scripts, as a best practice to avoid future issues.

7.8 Token Creators Can Avoid Fee Payments

Note Version 1

Token contracts that are native to the xDai side could be programmed such that they avoid a fee payment to the bridge validators, e.g. by simply ignoring transfer calls to and from the fee manager. Furthermore, existing tokens could be wrapped to avoid fees. However, as the fees are fairly low and as such tokens could be blocked on the bridge, the risk appears to be very low.

7.9 Token With Transfer Restrictions

Note Version 1

Certain Tokens, especially regulated stable coins, have transfer restrictions, blacklists or even the power to seize funds. If some tainted funds would be bridged, the entire bridge balance of that particular token might become frozen or could get seized. As with any other contract where funds are deposited, users need to be aware of these potential risks.

7.10 Weak Randomness

Note Version 1

The following function is used to pick a random number:

```
function random(uint256 _count) internal view returns (uint256) {
    return uint256(blockhash(block.number.sub(1))) % _count;
}
```

This is generally a bad way to sample randomness as, especially in the case of xDai, different attacks exist. Furthermore, there randomness is extremely slightly skewed. In this context, however, the randomness only serves to pick the account the receives the fee dust. As the corresponding monetary value is generally tiny, it seems acceptable.

7.11 onlyMediator Modifier

Note Version 1

There is an onlyMediator modifier inside the BasicAMBMediator contract. It performs two checks:

- Check that the call comes from AMB bridge contracts.
- Check that the forwarded by AMB bridge the message sender is a mediator on the other side.

There are multiple concerns about this modifier.

Firstly, the MediatorOwnableModule has a modifier with the same name that performs only one check - that the message comes from OmniBridge extension contract. That can potentially cause misunderstandings and human errors.

Secondly, it seems that the virtual message sender is always needed. This is currently being queried through a call to bridge.messageSender(). Here, for future versions of the AMB protocol a more efficient design would be possible where this information is passed along.

```
/**
 * @dev Throws if caller on the other side is not an associated mediator.
 */
modifier onlyMediator {
    _onlyMediator();
    _;
}

/**
 * @dev Internal function for reducing onlyMediator modifier bytecode overhead.
 */
function _onlyMediator() internal view {
    IAMB bridge = bridgeContract();
    require(msg.sender == address(bridge));
    require(bridge.messageSender() == mediatorContractOnOtherSide());
}
```