PUBLIC

Code Assessment

of the DAI Wormhole Smart Contracts

March 29, 2022

Produced for

MAKER

by

CHAINSECURITY

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 System Overview 6
4 Limitations and use of report 10
5 Terminology 11
6 Findings 12
7 Resolved Findings 13
8 Notes 17

1 Executive Summary

Dear Maker team,

Thank you for trusting us to help MakerDAO with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of DAI Wormhole according to Scope to support you in forming an opinion on their security risks.

The current version of the DAI Wormhole allows fast withdrawal (called "teleport" in the project's terminology) of DAI from a supported L2 solution onto L1 Ethereum. Using trusted oracles, the DAI can be issued on the receiver domain based on the promise that the amount will eventually be settled through the default bridge.

The most critical subjects covered in our audit are functional correctness and access control. Security regarding all the aforementioned subjects is high. General subjects covered were code complexity and gas efficiency. All the aforementioned subjects were of high quality.

In summary, we find that the codebase in its current state provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

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	1
• Code Corrected	1
Medium-Severity Findings	2
• Code Corrected	2
Low-Severity Findings	6
• Code Corrected	5
• Specification Changed	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 DAI Wormhole repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

DSS Wormhole

V	Date	Commit Hash	Note
1	21 December 2021	4db18d06cd5840d77acc16b916a851e1f5ccf137	Initial Version
2	14 February 2022	af7fc2b7c3a5a1e0ce79460793ae0f2bf3570c67	Second Version
3	18 March 2022	79fcb913c765a60ee86f72754da23238db076e63	Third Version

Optimism DAI Bridge

V	Date	Commit Hash	Note
1	21 December 2021	2c3fbbcce9dab9b6bea23eb57e941665069a50b7	Initial Version
2	14 February 2022	a03f13abb7f2197b95cd6c02f118c1c2b18e8a77	Second Version
3	18 March 2022	962ece1de4a49ccdb5928b081ee9e446f05d37b6	Third Version

Arbitrum DAI Bridge

V	Date	Commit Hash	Note
1	16 March 2022	f94ce7e72a391a4eec7970381c09736ab8fe4397	Initial Version

For the solidity smart contracts of dss-wormhole, initially the compiler version `0.8.9` was chosen. Optimization was not enabled during development. For the final version the compiler version was changed to `0.8.11` with optimization enabled. For the solidity smart contracts of optimism-dai-bridge, the compiler version `0.7.6` was chosen. For the solidity smart contracts of arbitrum-dai-bridge, the compiler version `0.6.11` was chosen.

The following files are in scope for this assessment:

dss-wormhole:

- WormholeConstantFee.sol
- WormholeFees.sol
- WormholeGUID.sol
- WormholeJoin.sol
- WormholeOracleAuth.sol
- WormholeRouter.sol
- relays/BasicRelay.sol

optimism-dai-bridge:

- common/WormholeGUID.sol

- l1/L1DAIWormholeBridge.sol
- l2/L2DAIWormholeBridge.sol

arbitrum-dai-bridge:

- common/WormholeGUID.sol
- l1/L1DaiWormholeGateway.sol
- l2/L2DaiWormholeGateway.sol

2.1.1 Excluded from scope

- The Maker Oracle Feeds are not part of this review. The oracleAttestations are fully trusted. This includes the certainty about finality of the individual L2 Domain, the resulting state changes of the respective L2 transaction and that the settlement of the DAI amount is guaranteed.
- An independent audit assesses the risks introduced by this new system.
- For the contracts in the optimism dai bridge repository only the new wormhole functionality was reviewed. The Optimism DAI bridge itself has been reviewed previously.
- For the contracts in the arbitrum dai bridge repository only the new wormhole functionality was reviewed. The Arbitrum DAI bridge itself has not been reviewed by ChainSecurity.
- Generally it is assumed that the message passing from L2 to L1 provided by the L2 solution works as documented and eventually relays the message onto L1.
- The EnumerableSet library from OpenZeppelin introduced in **Version 2** has not been reviewed.

3 System Overview

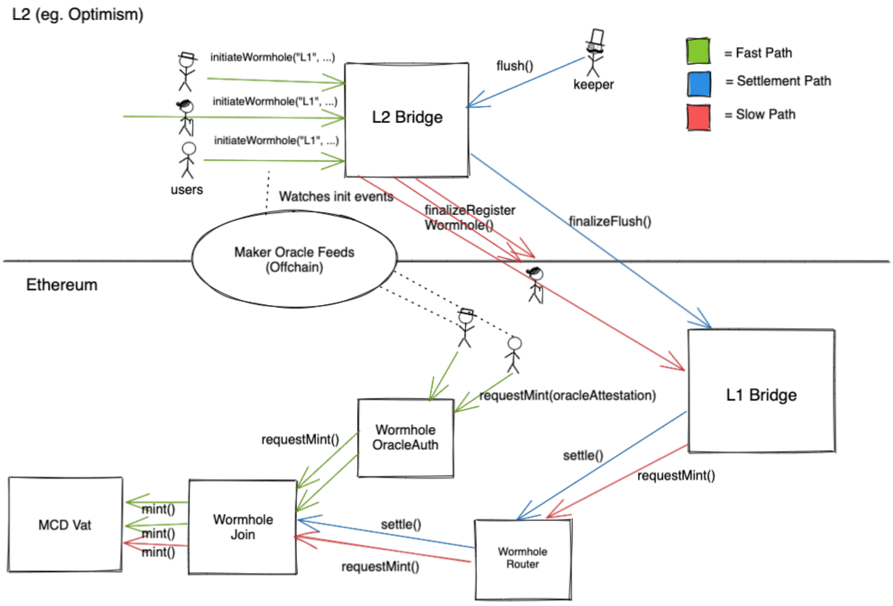
The current version of the DAI Wormhole allows fast withdrawal (called "teleport" in the project's terminology) of DAI from a supported L2 solution onto L1 Ethereum. In the future the DAI Wormhole will be generalized to support transfers between arbitrary L2 solutions / different chains (called "Domains"). Support for L2->L2 transfers of DAI will be added.

The most popular L2 solutions today all have a certain time delay for message passing from L2 to L1. Depending on the L2 solution this varies from a couple of hours up to 1 week. The DAI wormhole leverages special Maker Oracles monitoring the supported L2 solution's DAI Wormhole Bridge contract.

To initiate a fast transfer of DAI between different domains, the user calls `initWormhole()` on the L2DAIWormholeBridge contract. After it ascertained the finality of the transaction initiating the transfer the Maker Oracle issues a signed attestation allowing the user to immediately mint DAI on L1 through `WormholeOracleAuth.requestMint()` (green path).

From time to time, the collected DAI at the L2DAIWormholeBridge contract is transferred in one batched transaction via the normal bridge mechanism of this L2 solution onto L1 using `flush()`. After the message has been relied to L1, the transfer can be finalized using `finalizeFlush()` of the L1DAITokenWormholeBridge contract. On L1 the received DAI token are used to settle the debt created by the preliminary distributed DAI to the user (blue path).

As this approach described above requires the cooperation of the Maker Oracle Feeds, it is not trustless. In order remove the requirement to trust the Maker Oracles Feeds, the L2DAIWormholeBridge contract implements functionality allowing the users to transfer the message via the Bridge mechanism of the Domain / L2 solution. Note that depending on the L2 solution this bridge may not be trustless and requires to trust the sequencer. While this path is slower it allows the user to eventually retrieve his DAI (red path).



Fees: Transfers through the DAI Wormhole incur a fee which is taken when the DAI are minted on the destination domain. To determine the fee the code calls a fee adapter contract which may be changed. At the time of the audit one constant fee contract implementation existed.

Ilk: DAI must be available on L1 in order to immediately mint DAI to the user before the actual DAI tokens have been transferred via the bridge mechanism. A new `ilk` type has been introduced alongside the WormholeJoin adapter: Upon unwrapping the attestation issued by the trusted Maker Oracle Feed this promised DAI is locked as `gem` inside the VAT for an urn belonging to the WormholeJoin contract. Using this `gem` collateral, collateral is locked as `ink`, `art` and eventually DAI is generated. On settlement, the received DAI is joined back into the VAT and used to settle the debt. This special ilk has a fixed stability fee rate of 1.

Note that for L2->L2 transfers, while the teleporting happens directly the settlement takes a detour from L2(a) -> L1 -> L2(b).

3.1 Contracts:

- WormholeJoin: `join` contract representing the "promised Dai" `gem`, it is also responsible for managing the debt of the different domains
- WormholeRouter: the router facilitates the communication between gateways (WormholeJoin or bridges), it forwards minting requests and settlement from a gateway to another
- WormholeOracleAuth: this contract has the burden to receive and verify the signed messages emitted by the Maker Oracles and request the minting of Dai for the receiver. To be allowed to create the minting request, the message has to be signed by a certain number (`threshold`) of different Maker Oracles. `0` or `1` as `threshold` values have the same effect, i.e., only one approved signer is needed.
- WormholeFees: interface for wormhole fees
- WormholeConstantFees: implements WormholeFees with a constant fee
- WormholeGUID: contains the WormholeGUID struct, representing a minting request on a given domain, and allows to compute its hash
- L2DAIWormholeBridge (L2DaiWormholeGateway): wormhole bridge on Optimism (Arbitrum). Users will call `initiateWormhole`, this will burn Dai on L2, emit an event that will be picked by the

Maker Oracles (fast path) which will sign the WormholeGUID, and send a cross-domain message to let the user take the slow path if needed. This contract also has a `flush` function that will trigger the settlement on L1.

- L1DAIWormholeBridge: wormhole bridge on L1. The L2 cross-domain account can trigger its `finalizeFlush` function, which will trigger the settlement on L1. Any user can submit a WormholeGUID to the `finalizeRegisterWormhole` function, it will trigger a minting request (slow path).

3.2 Roles:

- Initiator: Account initiating the DAI transfer by calling `initiateWormhole()`. They can specify the `operator` and `receiver`.
- Operator: Account allowed to initiate the minting process on the destination domain.
- Receiver: Account receiving the minted DAI on the destination domain.

In this release the destination domain can only be L1 / Ethereum.

3.3 Trust Model

Wards: For every contract, each address set to `1` in any of the `wards` mapping is fully trusted and expected to behave correctly.

Maker Oracle Feeds: These actors and their Oracle attestations are fully trusted. The Oracle feeds are expected to sign the hash of a `WormholeGUID` if and only if they observe a `WormholeInitialized` event on L2. Furthermore, before signing they must ensure that the transaction is finalized and its state cannot change anymore. The authorized signers in `WormholeOracleAuth` are trusted to sign valid messages since they are the Maker Oracle Feeds.

The L2 bridges are trusted by Oracle Feeds and L1 bridges to emit minting events if and only if they receive a valid L2 withdrawal transaction.

Each gateway, either `WormholeJoin` or bridge deployed by Maker, is trusted by the `WormholeRouter` to be non-malicious.

Each contract in the scope of this audit is fully trusted by the other parts of the system to behave correctly at all time.

The users are not trusted and can only submit messages that have been signed by trusted entities, Maker Oracle Feeds or bridges.

Moreover, the Maker system needs to trust the L2 sequencer to include the wormhole transaction.

Changes in Version 2:

- The WormholeConstantFee implementation now only charges a fee for fast withdrawals: Withdrawals after `guid.timestamp + ttl` (`ttl is expected to be set to Time in seconds to finalize flush (not wormhole) via the slow path`) do not incur a fee.
- Users no longer specify a max fee but a max fee percentage based on the amount of the wormhole.
- Specifiying the `operator` for a wormhole is now optional.
    - The receiver can always mint DAI of the wormhole.
    - An `operatorFee` has been introduced. This is an additional fee in DAI and not subject to any limit checks (up to the full value of the wormhole), notably it's independent of the set `maxFeePercentage`.
- The fee interface has been changed to:

```
function getFee(
    WormholeGUID calldata wormholeGUID, uint256 line, int256 debt, uint256 pending, uint256 amtToTake
) external view returns (uint256 fees);
```

- `requestMint()` now returns the `postFeeAmount` and the `totalFee`.
- **BasicRelay**: This contract introduces a gasless relay for the oracle fast path. For wormholes with the operator set as this BasicRelay contract and given a valid signature from the receiver of the wormhole, anyone can relay the wormhole. The signature covers all parameters for the function call. `OracleAuth.requestMint()` is executed and succeeds only when the full amount of the wormhole has been successfully minted, the `gasFee` amount of DAI is transferred to the relayer.

Changes in Version 3:

- `WormholeJoin.cure()` now returns the total debt used by this contract based on the stored `art` value. Only actions of WormholeJoin can modify this value, and the value stored is always updated accordingly. The change is necessary in order for the function to return the debt correctly under all circumstances, notably during shutdown of the VAT where `end.skim()` may "settle" the vault.

In version 3 of the optimism dai bridge contracts the DAIWormholeBridge contracts (L1 and L2) have been renamed to DAIWormholeGateway.

4 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 the 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 defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. 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 that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

5 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 severity. 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.

6 Findings

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

- 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	0
Low-Severity Findings	0

7 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	1

- Fees May Block Slow Path Code Corrected

Medium-Severity Findings	2

- L2 Addresses Code Corrected
- Minting Pending DAI Incurs Additional Fees Code Corrected

Low-Severity Findings	6

- Missing or Incomplete Natspec Code Corrected
- Specification Mismatch Specification Changed
- Code Inefficiency Code Corrected
- Interface Mismatch Code Corrected
- Missing Index Code Corrected
- file() Casting Bytes32 to Uint256 Code Corrected

7.1 Fees May Block Slow Path

Design High Version 1 Code Corrected

The slow path goes through `L1DAIWormholeBridge.finalizeRegisterWormhole()` which calls `requestMint` with `maxFee = 0`.

When the `vat` is live, the computed fee in `_withdraw` function of `WormholeJoin` may be `> 0` and the transaction would revert due to:

```
require(fee <= maxFee, "WormholeJoin/max-fee-exceed");
```

This essentially prevents users who are censored by the oracle to redeem using the slow path.

Code corrected:

The only fee currently present, the WormholeConstantFee, now features a ttl after which the fee returned for this WormholeGUID is 0.

7.2 L2 Addresses

Design Medium Version 1 Code Corrected

The address format can differ across L2 systems / different domains the DAI wormhole connects. While the majority work with address of 20 bytes, compatible with the solidity `address` type, other systems can use other address format. One example of those is StarkNet where addresses of are of type `felt` which are larger than 20 bytes.

Code corrected:

The `receiver` and `operator` fields of the `WormholeGUID` struct have been replaced by `bytes32` types to accommodate for address formats up to 32 bytes.

7.3 Minting Pending DAI Incurs Additional Fees

Design | Medium | Version 1 | Code Corrected

Using the DAI Wormhole may take a fee from the user. This fee is taken on L1 and transferred to the VOW. The fee is accounted for inside `_withdraw()` and calculated using an external Fee adapter contract based on the `wormholeGUID` which contains all information about the transfer, the current debt and the line, the debt ceiling according to the source domain.

The amount of the fee taken is calculated before determination of the amount that is withdrawn.

```
uint256 fee = vatLive ? FeesLike(fees[wormholeGUID.sourceDomain]).getFees(wormholeGUID, line_, debt_) : 0;
require(fee <= maxFee, "WormholeJoin/max-fee-exceed");

uint256 amtToTake = _min(
                        pending,
                        uint256(int256(line_) - debt_)
                    );
```

The fee is based on the full amount of the wormholeGUID being processed, not on the actual amount withdrawn in this transaction. The actual amount withdrawn is limited by the maximum debt that can be created without exceeding the ceiling. The remaining amount can be retrieved later when more debt can be accrued using `mintPending()`. This however again uses function `_withdraw` which again calculates the fee based on the full amount of the wormholeGUID, the current debt and debt ceiling. The pending amount is not taken into account for the calculation of the fee.

Hence, should the amount to be withdrawn be limited by the remaining space between the debt ceiling (line) and the current debt, the user pays fees based on the full amount, not the amount being withdrawn. Later, when the remaining pending amount is withdrawn, the user again pays fees based on the full amount of the wormholeGUID, effectively paying again for the same transfer.

Code corrected:

The fee computation function `getFee` takes more parameters (`pending`, `amtToTake`) into account. This allows more versatile ways to compute the fee. For example, `WormholeConstantFee` can now compute the fee relative to the amount being withdrawn instead of the full fee every time the full amount is partially withdrawn.

7.4 Missing or Incomplete Natspec

Design | Low | Version 2 | Code Corrected

- `requestMint`, `_mint` and `mintPending` are missing the natspec for their second return value `totalFee`.

- the `cure` and `getFee` functions specification should specify the unit of its return value.
- the `isValid` function specification should describe the return value.
- the `v`, `r` and `s` parameters of `BasicRelay.relay` should be described in the specification

---

Code corrected:

The issues raised above have been addressed.

7.5 Specification Mismatch

Correctness | Low | Version 2 | Specification Changed

The specification of the `mintPending` function says that it is only callable by the operator, but the receiver is also allowed to call the function.

Specification changed:

The description of the `mintPending` function has been fixed.

7.6 Code Inefficiency

Design | Low | Version 1 | Code Corrected

- `signers` mapping in `WormholeOracleAuth` is `address => bool`, it would be more gas efficient to have a `address => uint256`.
- in `WormholeOracleAuth`, `threshold` is passed as a function parameter in `isValid`. `threshold` is a storage variable in the contract and thus can be accessed directly from `isValid` function and does not need to be passed as a parameter, this would save gas.
- in `_withdraw` function of `WormholeJoin`, the overflow check for `_line` happens every time. Checking for overflow only once in the `file` function where the line for a domain is set would save gas.

Code corrected:

- the `signers` mapping has been changed to a `mapping(address => uint256)`.
- MakerDAO wants the `isValid` function to be used by anyone who wants to verify an oracle attestation.
- the `file` function checks for `_line` validity and the check has been removed from `_mint` (new version of `_withdraw`).

7.7 Interface Mismatch

Design | Low | Version 1 | Code Corrected

- The signature of the DAI token's `approve` is `function approve(address, uint256) external returns(bool);`, but `WormholeJoin` and `L1DAIWormholeBridge` have it as `function approve(address, uint256) external;` in the interface they define for `TokenLike`.

Version 2:

- The interface signature of `requestMint` in WormholeRouter exposes only one return value out of two. The interface signature of `requestMint` in L1DAIWormholeBridge exposes no return value at all. The compiler will just drop the unused return values without causing an error, but this design choice does not reflect the correct signatures and should be documented.

Code corrected:

- `WormholeJoin` and `L1DAIWormholeBridge` have the correct interface for the DAI token's `approve` function.
- The interface signature of `requestMint()` in WormholeRouter has been fixed in Version 3.
- L1DAIWormholeBridge is now called L1DAIWormholeGateway. The interface is now defined correctly in the imported WormholeInterface.

7.8 Missing Index

Design Low Version 1 Code Corrected

- `domain` fields are indexed in `WormholeJoin` events. It could be useful to index the `domain` field of `WormholeRouter`'s `File` event to make it more easily searchable.
- `targetDomain` field in `Flushed` event of `L2DAIWormholeBridge` can be indexed to ease its search.

Code corrected:

- the `domain` fields in the `File` events are indexed.
- the `targetDomain` field in the `Flushed` event is indexed.

7.9 `file()` Casting Bytes32 to Uint256

Design Low Version 1 Code Corrected

Contrary to the other contracts which have multiple `file` functions with the `data` parameter of the actual type of the data passed, the WormholeOracleAuth has a `file` function taking a `bytes32` argument as data which is then casted to `uint256`.

Code corrected:

The `file` function responsible for the `threshold` parameter now takes directly a `uint256 data` to avoid an unnecessary conversion.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Breaking Changes of the Solidity Compiler

Note **Version 1**

The new compiler version behaves differently on code related to integer conversion/negation when the value is exactly 2**255.

The core DSS system which has been compiled with compiler version 0.6.12 first ensures that the value of the uint is below or equal to 2**255 before converting it to a (negative) integer.

An example of this pattern can be found in e.g. `GemJoin.exit()`:

```
require(wad <= 2 ** 255, "GemJoin/overflow");
        vat.slip(ilk, msg.sender, -int(wad));
```

This project, DSS-Wormhole uses a more recent compiler version 0.8.9. Negating 2**255 is no longer possible and will result in the transaction reverting.

In DSS-Wormhole, `WormholeJoin.settle()` features such a pattern:

```
function settle(bytes32 sourceDomain, uint256 batchedDaiToFlush) external {
    require(batchedDaiToFlush <= 2 ** 255, "WormholeJoin/overflow");
    daiJoin.join(address(this), batchedDaiToFlush);
    if (vat.live() == 1) {
        (, uint256 art) = vat.urns(ilk, address(this)); // rate == RAY => normalized debt == actual debt
        uint256 amtToPayBack = _min(batchedDaiToFlush, art);
        vat.frob(ilk, address(this), address(this), address(this), -int256(amtToPayBack), -int256(amtToPayBack));
```

Note that in this case the check is superflous: The multiplication in daiJoin.join() will revert due to an overflow on even lower values.

Nevertheless it's important to be aware of this behavior, the same code pattern behaves differently depending on the compiler version. This requires careful attention especially when such contracts, which have been compiled with different compiler versions, e.g. DSS-Wormhole and the VAT interact.

8.2 Finality and State Change on L2

Note **Version 1**

The notion of finality of transactions and the resulting state change differs across the L2 solutions. The Wormhole system, especially the Maker Oracle Feeds must be aware of that and take each finality definition into account. Ideally this is properly assessed and documented for each Domain the Wormhole connects to.

8.3 Slow Path Requires Zero Fee

Note **Version 1**

The slow path through `L1DAIWormholeBridge.finalizeRegisterWormhole()` requires successful redemption of the wormhole with no fee due to:

```
function finalizeRegisterWormhole(WormholeGUID calldata wormhole)
    external
    onlyFromCrossDomainAccount(l2DAIWormholeBridge)
{
    wormholeRouter.requestMint(wormhole, 0, 0);
}
```

The interface definition of `WormholeFees` emphasizes this:

```
It should return 0 for wormholes that are being slow withdrawn.
```

8.4 Surplus DAI for WormholeJoin in the VAT

Note **Version 1**

Function `settle()` of the WormholeJoin contract is permissionless and given enough DAI token balance of the WormholeJoin contract (e.g., provided by the caller) can be executed by anyone. `DaiJoin.join()` returns the DAI tokens into the system and the contract's balance tracked by the `DAI` mapping of the VAT increases accordingly. This increased balance however is stuck when everything has been settled.

Note that the `sourceDomain` can be chosen arbitrarily by the untrusted caller. Listeners of the event `Settle` must be aware that this event may be triggered by anyone and may not represent a debt repayment coming from the bridges.