PUBLIC

Code Assessment

of the StarkNet-DAI-Bridge Smart Contracts

December 6, 2021

Produced for

by

Contents

1 Executive Summary

Dear all,

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 StarkNet-DAI-Bridge according to Scope to support you in forming an opinion on their security risks.

MakerDAO implements a layer 2 DAI contract for StarkNet, a ZK-Rollup for Ethereum, and DAI bridging contracts from the layer 1 to layer 2. That also includes contracts for sending governance spells from layer 1 to layer 2.

The most critical subjects covered in our audit are the functional correctness and security of the DAI bridging mechanism, the functional correctness of the L2-DAI ERC-20 contract, the protection against censorship, and the functional correctness of relaying governance spells.

The documentation of the project contains a risk section discussing potential threats which helps the overall security of the project.

The security and the functional correctness of the reviewed version of the smart contracts is high, all critical and high severity issues uncovered in previous iterations of the review have been fixed.

In summary, we find that the codebase 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. Furthermore, due to the experimental nature of the L2 solution some risks remain.

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	1
• Code Corrected	1
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	4
• Code Corrected	4
Low-Severity Findings	5
• Code Corrected	5

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 StarkNet-DAI-Bridge 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	22 November 2021	ba740acd782c18650ad0c70d623ba3b03087d49e	Initial Version
2	2 December 2021	14c52313b10e8006e66835690ed5eb6546be2b5a	After Intermediate Report
3	6 December 2021	bdea59fd2d273bf7a91f3dc5f745ba2a66560805	Final Version

For the solidity smart contracts, the compiler version `0.7.6` was chosen. For the cairo smart contracts, the compiler version `0.6.0` was chosen.

2.1.1 Excluded from scope

The contracts `l1/mocks.sol`, `l2/account.cairo` and `l2/registry.cairo` are not part of this review. For the purpose of this audit, the StarkNet bridge and the corresponding contracts are assumed to work correctly. They are not part of this review. For the cairo contracts, the focus was on the source code files inside the repository. The imported functionality from `starkware.starknet.common.*`, `starkware.cairo.common.*` as well as the built-ins have not been reviewed in depth, generally these are assumed to work as described.

StarkNet Alpha has been released recently and has not been audited yet. StarkWare states that changes, fixes and improvements are to be expected. This review took place at the end of November 2021 and cannot account for future changes and possible bugs in StarkNet. Amongst others, most notably at the time of audit the fee model was not yet specified.

2.2 System Overview

The set of Solidity and Cairo contracts implement a bridge for DAI from Ethereum on layer 1 to the StarkNet layer 2 solution and vice versa. It follows the concept of other similar DAI bridges from L1 to other L2 solutions.

Following contracts are deployed:

- On L1:
    - **L1DAIBridge**
    - **L1Escrow**
    - **L1GovernanceRelay**

- On L2:
    - **dai**
    - **l2_dai_bridge**
    - **l2_governance_relay**

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

2.2.1 L2 DAI

On the L2 side, a DAI contract with functionality similar to an ERC20 token is deployed. All default ERC20 functionality such as `transfer`, `transferFrom`, `approve`, `allowance`, `balanceOf` and `totalSupply` are implemented. Note as events do not exist on StarkNet, the token contract does not emit any of the default ERC20 events. Additionally the DAI contract implements `mint` which is used by the token bridge to mint tokens after a deposit from L1. The `burn` function which can be used by anyone to burn his tokens is used by the bridge contract during withdrawal of DAI from L2 back to L1. In order to mitigate the approval problem of the default ERC20 approve function, the contract implements `increase_allowance()` and `decrease_allowance()`. Finally, a magic number used as amount during approval works as truly unlimited approval, for callers with this magic number as approved amount the approved amount is not reduced.

On StarkNet transactions sent into the network do not have a defined sender. External parties sending transactions into the network cannot be identified by an address. Hence in order to hold tokens on StarkNet, each user must deploy a contract which has a unique address. Such an address can then be used in the token contract to hold a balance and/or a privileged role.

2.2.2 Bridge

The L1DAIBridge contract implements all logic to deposit and withdraw DAI to and from L2. The deposited DAI are held by the L1Escrow contract. This separation of the funds from the logic makes the system upgradable: the L1DAIBridge logic contract can be replaced independently of the Escrow holding the DAI.

Similarly, the l2_dai_bridge contract implements the logic for handling deposits and withdrawals. Note that on L2 the bridge can mint and burn DAI and no escrow is needed on L2. Also, the bridge on L2 can be upgraded through a governance spell through the l2_governance_relay contract that could remove the bridge's minting and burning rights and assign them to a new bridge.

The bridge may be paused independently on L1 and L2. Once a bridge contract is paused, it cannot be unpaused. Pausing the L1 bridge contract means deposits from L1 are no longer possible. Pausing the L2 bridge contract prevents new withdrawals from being initiated. Finalizing withdrawals on L1 that have been initiated before the L2Bridge contract was paused remains possible even when the L1 bridge contract is paused. Similarly, even when the L2 bridge contract is paused, deposit requests from L1 can still be finalized. The forced withdrawal functionality works when the L1 bridge is not paused and is independent of the pause state of the L2 bridge.

Furthermore, the contract features a ceiling limiting the total amount of DAI that can be deposited using this bridge. That is a protection to limit the impact of a potential bug on the core Maker system. The ceiling may be changed while the system is live.

Depositing DAI from L1 works as follows:

1. First the user approves the L1DAIBridge contract to transfer the amount of DAI.
2. Next the user calls L1DAIBridge.deposit() passing the amount, the source of the funds and the destination address for the DAI on L2.
3. After a period of blocks the StarkNet bridge automatically process the transaction on L2 and mints the DAI for the specified address

Withdrawing DAI from L2 works as follows:

1. The user executes `withdraw()` on the l2_Dai_bridge contract. He specifies the amount and the L1 address that will be able to claim the funds on L1.
2. After a period of time the StarkNet bridge has made the message available for consumption on L1. Now the specified caller can call `L1DAIBridge.finalizeWithdrawal()` using the correct parameters and the destination where the DAIs should be transferred to.

A user may have multiple unconsumed withdrawals pending at the same time.

2.2.3 Governance Relay

The governance relay contract on L2 is a ward, a privileged account, in all L2 contracts of the system. The MakerDAO governance on L1 can decide to execute any action on L2 using this contract. Technically this works as follows: A new contract is deployed on L2 featuring code to execute. On L1 the governance decides to execute this action and the call is relayed via the L1GovernanceRelay contract. On L2 this triggers the execution of the specified contract's code as delegatecall in the contract of the l2_governance_relay contract.

2.2.4 ForceWithdrawal

Currently StarkNet is centralized. Transactions on L2 are executed by a so called sequencer, which may censor transactions on L2 or could become unavailable for other reasons. This may block users from transferring DAI on L2 or initiating withdrawal of the DAI to L1. The L1DAIBridge contract features a `forceWithdrawal` function. This function documents the withdrawal attempt on L1 and attempts to force withdraw the DAI from L2. The sequencer may or may not execute this operation on L2. It's important to understand that the prover cannot prove a failed execution. This means reverting transactions are indistinguishable from censored messages. Hence `finalize_force_withdrawal` must prevent all possible reverts e.g. due to insufficient balance/allowance and ensure the transaction terminates gracefully so it can be executed and proven. Either the withdrawal is executed or the failed attempt is visible on L1 which would document such a behavior of the sequencer and allow the DAO Governance to shut down the bridge and release the DAI of the users.

The process of the force withdrawal may be described as follows:

1. User initiates a force withdrawal on L1.
2. A message to L1 is sent. The proof of sending the message is set in the StarkNet L1 contract.
3. On proven consumption of the L1 message on L2, the message sent from L1 is marked as consumed.
4. L2 sends a withdrawing message to L1.
5. User can withdraw his funds.

Note that failed consumption of a message can be monitored by the value set in 2. Disallowing any reverts in the l1_handler in the L2 bridge contract allows for a distinguishment of censorship / system failure and failed messages.

On L2, a registry contract exists which allows user on L2 to specify their address on L1 for the forced withdrawal functionality. For the forced withdrawal to work successfully on L2 the user must have this address set as this is used for access control.

Furthermore, should the MakerDAO governance assisted evacuation procedure be initiated as the Layer 2 solution starts censoring or becomes unavailable this registry is used to match L2 to L1 addresses during reimbursement. Ideally, users register their L1 address before receiving DAI on L2.

2.3 Trust Model & Roles

We assume the deployment and initialization (e.g. of the ward roles in order for the L1DAIBridge to be able to transfer DAI from the Escrow during withdrawal, for the l2_dai_bridge to be able to mint DAI on L2 or the GovernanceRelay) to be done correctly before the bridge becomes operational.

- **User**: Users are fully untrusted.
- **Wards**: Accounts holding the ward role in a contract of the maker system have access to all privileges functionality of this contract. As required the other contracts of the system and the Maker governance have the ward role in the contracts. These parties are assumed to act honestly and correctly at all times.
- **StarkWare**: Partially trusted. Operates the bridge from L1 to L2, executes the L2 transactions. Generally trusted to work without censorship, however, the smart contract implements a fallback should the party censor transactions on L2 or become unavailable
- **Registry**: Trusted external contract. Note that the force withdraw mechanism relies on the security and correctness of this contract.

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	0
Low-Severity Findings	0

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	1

- Unprotected Escrow Funds Code Corrected

High-Severity Findings	1

- L2 DAI Allows Stealing Code Corrected

Medium-Severity Findings	4

- ForceWithdrawal Needs Prior Approval Code Corrected Specification Changed
- L2 Address Sanity Checks Code Corrected
- Relay Parameter Mismatch Code Corrected
- Unlimited Approvals and the Range of Uint256 Code Corrected

Low-Severity Findings	5

- ERC-20 Functions Have No Return Values Code Corrected
- Inconsistent Version Pragma Code Corrected
- Inefficiency in Reading Allowances Code Corrected
- Lack of L1-address Sanity Checks on L2 Code Corrected
- Unused Code Code Corrected

6.1 Unprotected Escrow Funds

Security Critical Version 1 Code Corrected

`L1DAIBridge.deposit()` transfers DAI from a user-specified address `from` to the `L1Escrow` contract to lock DAI on layer one. However, a malicious user could specify `from` to be the `L1Escrow` contract that holds all of the locked funds. The call to `DAI.transferFrom()` will succeed since the escrow must have had approved the bridge contract. Ultimately, unbacked DAI could be minted on L2 and funds from the escrow could be stolen.

Consider the following scenario:

1. User calls `deposit()` with `from` being the escrow contract.
2. The self-transfer from and to escrow succeeds as long as `amount <= allowance[escrow][bridge]`.
3. The ceiling check passes as long as `balanceOf(escrow) <= ceiling` since the balance does not change.
4. Ultimately, a message to L2 is sent and unbacked DAI on L2 is minted.
5. Repeat the process.
6. Withdraw DAI from L2 to L1, such that the escrow is emptied.

The README.md file in the repository states:

```
### Initial configuration

    ... Unlimited allowance on `L1Escrow` should be given to `L1DAIBridge`.
```

Hence an attacker may drain all DAI out of the escrow.

Furthermore, e.g. by frontrunning a `deposit` transaction or exploiting an unlimited approval given by the user to the bridge it is possible to steal L1 DAI from users. Consider the following scenario:

1. User A intends to deposit DAI to L2 and approves the bridge contract. He either gives an exact approval for the amount he wants to deposit or may give an unlimited approval as he trusts the bridge contract and intends to use it in the future. Next he crafts a transaction to `deposit`.
2. User B calls `deposit()` and specifies the `from` address to be user A. The call succeeds and B receives funds on L2. Note that the DAI locked on L1 are from user A. This transaction frontruns the `deposit` call coming from user A.
3. User A's deposit is executed but fails due to lack of allowance.

Note that although they are known to be potentially dangerous it is quiet common that users give infinite approval to such systems they trust and intend to interact with frequently.

Code corrected:

The `from` parameter has been removed from function `deposit`. The DAI amount is now transferred from `msg.sender` to the escrow. Hence the issue described above no longer exists.

6.2 L2 DAI Allows Stealing

Security | High | Version 1 | Code Corrected

The `transfer` function of the L2 DAI contract allows stealing tokens from other users. The attack works as follows:

1. Within the `amount` field of the `transfer` function the user specifies an invalid `Uint256`. Note that `uint256_check` is never called. To steal `i` token wei, the attacker specifies `P-i` to be `amount.low` and `0` to be `amount.high`. The low amount could be interpreted as the negative number `-i`.
2. The `uint256_le(amount, sender_balance)` check will be passed as it will ultimately compute the following:

   ```
   1 - is_nn(amount.low - (sender_balance.low+1))
   ```

   If for example the sender's (attacker's balance) is 0, that check will pass.
3. The `uint256_sub(sender_balance, amount)` computation will result in an increased `sender_balance` due to the specially crafted `amount`.
4. The `uint256_add(recipient_balance, amount)` computation will result in a decreased `recipient_balance` due to the specially crafted `amount`.

Note that the decrease of the `recipient_balance` is also the increase of the `sender_balance`. In other words, the sender gains as many tokens as the recipient loses. Or more concisely, the sender can steal all of the tokens of the receiver. So, if `i==1` then one token wei is stolen. If `i==2` then two wei are stolen.

The only precondition for the attack is that the `uint256_le(amount, sender_balance)` can be passed for manipulated `amount` values. Note that the current hints prevent a proof generation for this attack in `uint256_add`, but hints can freely be changed and the verifier will accept it.

Code corrected:

`amount` is now validated in the internal function `_transfer`. Thus, neither `transfer()` nor `transfer_from` can perform computations with invalid integers. Ultimately, the `Uint256` library functions receive the expected inputs and, thus, perform the documented computations.

6.3 ForceWithdrawal Needs Prior Approval

Correctness | Medium | Version 1 | Code Corrected | Specification Changed

In the case that a user believes they are censored, the user can initiate the withdrawal using the `forceWithdraw` function of the L1DAIBridge. When the L2 network works as expected, the withdrawal request is handled.

This however has some prerequisites:

1. The user needs to have registered his L1 address in the L2 registry prior to initiating `forceWithdraw()`. Note that this may no longer be possible when the L2 network is censoring transactions hence this should be done by the users before receiving DAI on L2.
2. The execution of `finalize_force_withdrawal` on L2 in case the Layer2 network complies requires that the user has previously given allowance to the `l2_dai_bridge`. Again, giving the approval at this point in time may no longer be possible in case the L2 network censors transactions.

```
# check allowance
let (contract_address) = get_contract_address()
let (allowance : Uint256) = IDAI.allowance(dai, source, contract_address)
let (allowance_check) = uint256_le(amount, allowance)
if allowance_check == 0:
    return ()
end
```

This requirement is not documented and may come as a surprise for the user. Note that for normal withdrawals from L2 using `withdraw` no such allowance is needed. Furthermore without the check in `finalize_force_withdrawal` the withdrawal / burning of the DAI would work as the `l2_dai_bridge` is a `ward` in the DAI contract and has the privilege to burn the DAI of any address without the need for an approval.

The case that the L2 network may only censors transactions other than forced withdrawals (in order to avoid detection of the misbehavior) and its implication must be considered.

Overall the ForcedWithdrawal process and it's restrictions is not documented enough.

Code corrected and specification changed:

Issue 1) was addressed by improving the documentation. The documentation now clearly states what actions are required before a forced withdrawal can be executed. The enhanced documentation also resolves 2), note that in the updated code a `ward` of the DAI contract no longer has the privilege to burn DAI and hence the approval is needed. It's important to understand why `finalize_force_withdrawal` must check whether the approval exists: Burning without the allowance would result in the transaction to revert. The prover can't prove failed executions, reverts are

indistinguishable from censored messages. By checking the allowance and gracefully terminate the transaction when no sufficient allowance exist, the transaction can be executed. Hence the message from L1 can be processed which allows to clear the message in the StarkNet contract on Ethereum. This proves that the transaction must have been executed on L2.

6.4 L2 Address Sanity Checks

Design | Medium | Version 1 | Code Corrected

In StarkNet users do not have addresses. Transactions sent to the network have the 0 address as caller. In order to identify accounts via addresses, each user deploys his account contract and interacts with contracts such as the DAI token using his account-contract.

- The `deposit()` function of the L1DAIBridge contract allows users to deposit with the `to` address set to 0. The execution of `finalize_deposit` initiated by the l1_handler on l2 however will fail as minting DAI for the zero address will revert. As a result the deposited DAIs on L1 will be locked in the escrow.

Furthermore, note that `to` will be received as a `felt` on L2. Hence, the true `to` address on L2 will be `to % R`. Therefore, it could be possible to for example specify address `R` on L1 which will map to zero-address (similarly `R+1` will map to address 1). Users could be protected from errors by restricting the allowed address range on L1.

- L2 DAI allows to give approvals specifying the 0 address as caller. All holders of L2 DAI must be aware that this is very dangerous and means that anyone crafting an external transaction to the network can transfer their DAI using this approval.
- A user could specify the l2_dai contract as the recipient of the funds on `deposit`. Since the L1 call would succeed while the L2 call to the l1_handler would fail, the cross-layer message would remain unconsumed.

Code corrected:

The code does the following checks now on L1:

- `to != 0` to ensure that the address is non-zero.
- `to != l2Dai` to prevent a failing mint.
- `to < SN_PRIME` to prevent a possible StarkNet overflow.
- All functions related to approvals in the l2 DAI contract now forbid approving the zero-address.

6.5 Relay Parameter Mismatch

Correctness | Medium | Version 1 | Code Corrected

The L1GovernanceRelay is used to send messages to the L2 GovernanceRelay to execute spells. However, the parameters sent by the L1 contract and the parameters the L2 contract receives do not match. Ultimately, governance spells cannot be relayed to L2.

More specifically, the L1GovernanceRelay sends a message to L2 as follows:

```
uint256[] memory payload = new uint256[](2);
payload[0] = to;
payload[1] = selector;
```

```
StarkNetLike(starkNet).sendMessageToL2(l2GovernanceRelay, RELAY_SELECTOR, payload);
```

However, the L2 side of the governance relay consumes the message as follows:

```
@l1_handler
func relay{
  syscall_ptr : felt*,
  pedersen_ptr : HashBuiltin*,
  range_check_ptr
 }(
   from_address : felt,
   target : felt
 ):
   let (l1_governance_relay) = _l1_governance_relay.read()
   assert l1_governance_relay = from_address
   let (calldata : felt*) = alloc()
   delegate_call(target, EXECUTE_SELECTOR, 0, calldata)


  return ()
 end
```

The arguments of the L1 handler should consist of the `from_address` and `payload`. However, the `payload` created on L1 has two elements. That ultimately lets the execution of a governance spell fail.

Code corrected:

The unused selector was removed from the payload, the payload now contains the spell only.

6.6 Unlimited Approvals and the Range of Uint256

Design | Medium | Version 1 | Code Corrected

DAI on L1 supports unlimited approvals using `uint256(-1)` as magic value. When an approval for this magic value is given, the spender can spend the funds of the token holder without the allowance being reduced.

Similarly the DAI contract in cairo supports an unlimited approval using a different magic number. As Uint256 work differently in cairo, it's possible to define a magic value outside the actual range of Uint256. In cairo, a Uint256 is represented by a struct containing two `felt` members:

```
struct Uint256:
    # The low 128 bits of the value.
    member low : felt
    # The high 128 bits of the value.
    member high : felt
end
```

However note that a `felt` can store more than 128 bits, so a Uint256 represented by such a struct may contain a value exceeding the max uint256 value.

The code of the DAI cairo contract, however, takes advantage of this special property of the Uint256 type and defines the magic number for the unlimited approval as:

```
const MAX_SPLIT = 2**128
let MAX = Uint256(low=MAX_SPLIT, high=MAX_SPLIT)
```

Note that the common library for Uint256 offers a function `uint256_check` which checks if the given Uint256 is actually valid. The code of the DAI cairo contract uses this function to check whether amounts regarding balances are valid. In contrast, the code is generally not using `uint256_check()` when handling or checking approvals. That results in following potentially intended and/or strange behaviour:

- Function `approve` can be used to give allowance for a valid amount, the magic number or an invalid uint256 value.
- Function `increase_allowance` does not work on such allowances due to the carry over. However, increasing with bad input values could decrease the allowance (in a similar fashion as described in L2 DAI allows stealing).
- Function `decrease_allowance` works. However, note that decreasing to the magic number results in unlimited approval so that allowance has been increased instead of decreased.

Concluding, the selection of the magic value outside the valid range for Uint256 could lead to unexpected and undocumented behaviour due to an implied lack of Uint256 validity checks. Furthermore, the deviation from L1-DAI's magic value may confuse users.

Code corrected:

`MAX_SPLIT` has been renamed to `ALL_ONES` and redefined to `2**128-1`. Also, `uint256_check()` is called now in the functions `approve`, `increase_allowance` and `decrease_allowance`. Since the inputs are always validated and allowance cannot be out of the valid `Uint256` range, the unintended behaviour cannot occur anymore.

6.7 ERC-20 Functions Have No Return Values

Design | Low | Version 1 | Code Corrected

EIP-20 specifies that for example `transfer` has a boolean return value. However, L2 DAI does not return anything.

Code corrected:

Return values have been implemented for the ERC-20 functions.

6.8 Inconsistent Version Pragma

Design | Low | Version 1 | Code Corrected

Different to the L1Escrow contract, the L1DAIBridge and the L1GovernanceRelay contract feature following version pragma:

This allows the contracts to be compiled with any Solidity version >= 0.7.6 including more recent major version which may feature changes in the syntax.

The Solidity documentation states:

```
Source files can (and should) be annotated with a version pragma to reject
compilation with future compiler versions that might introduce incompatible changes.
```

For more information, please refer to: https://docs.soliditylang.org/en/develop/layout-of-source-files.html#version-pragma

---

Code corrected:

The pragmas have been changed to:

```
pragma solidity ^0.7.6;
```

6.9 Inefficiency in Reading Allowances

Design | Low | Version 1 | Code Corrected

In function `burn` of the DAI cairo contract the allowance is always read. However, it is only used if `check_allowances == 1` is `true`. Thus, the efficiency of the functionality could be improved. Similarly, that is the case for `transferFrom()`.

Code corrected:

In the updated code wards no longer have special privileges in `dai.burn()`. Due to the changed code, the issue described above no longer applies.

6.10 Lack of L1-address Sanity Checks on L2

Design | Low | Version 1 | Code Corrected

L1 addresses on L2 are of `felt` type. However, that could ultimately lead to bad user-input on L2 when passing L1 addresses since L1 addresses have 160 bits which is less than the number of bits the `felt` type is represented with.

For example, in function `withdraw()` of the L2 bridge contract a user passes an L1 address as `felt` which could to a bad address being passed to L1.

Code corrected:

A check has been added to `send_finalize_withdraw()` with ensures that the destination is a valid L1 address. This function is used by both, `withdraw` and `finalize_force_withdrawal`.

In the initial round of fixes the `assert_l1_address` function contained unnecessary declarations of `local syscall_ptr` and `local pedersen_ptr` which now have been removed.

6.11 Unused Code

Design | Low | Version 1 | Code Corrected

The L1DAIBridge contract defines the `struct SplitUint256`. However, it remains unused.

Code corrected:

The unused struct was removed.