Formal Verification of Origin's OUSD Token

Summary

This document describes the specification and verification of Origin's OUSD Token using Certora Prover. The work was undertaken from January 10th 2021 until January 15th 2021. The latest commit that was reviewed and run through the Certora Prover was 5c702edbefb451287805918c3275535c11654074.

The scope of our verification was the OUSD Token contract (`OUSD.sol`).

The Certora Prover proved the implementation of the OUSD token is correct with respect to the formal rules written by the Origin and the Certora teams. During the verification process, the Certora Prover discovered minor bugs in the code listed in the table below. All issues were promptly corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The Certora development team is currently handling these limitations. The next section formally defines high level specifications of OUSD.

All the rules are publically available in a public github:
https://github.com/OriginProtocol/origin-dollar/tree/certora-audit

List of Main Issues Discovered

Issue: 1 OUSD unit could be worth more than a $1.
Rule(s) broken: Solvency.
Description: The total value stored in the vault is computing the sum of the vault's and its strategies holdings' in a basket of stablecoins. However, there could be (hopefully small) fluctuations in the values of these stablecoins, therefore rebasing a certain amount of OUSD based on this sum could lead to a bigger available supply of OUSD then is actually covered by the vault.
Severity: High.
Mitigation: Achieving this is strategy-dependent. Ensure strategies only increase the value they return.

Issue: Governor could mistakenly pull out tokens used to back OUSD.
Rule(s) broken: Solvency.
Description: The governor could invoke the escape hatch for ERC20 tokens on assets and amounts that are stored in the vault for backing OUSD.
Severity: Medium.
Mitigation: Fixed - allowing transfer of only non-supported assets.

Issue: totalSupply can exceed max supply.
Rule(s) broken: total supply of OUSD must be below max supply.
Description: The computation in changeSupply rounds up the resulting totalSupply, so eventually it is theoretically possible to exceed the max supply.
Severity: Medium.
Mitigation: Fixed - Added a require in changeSupply.

Issue: burn operation is not additive.
Rule(s) broken: burn is additive.
Description: Burning in parts may lead to greater balance in the end then burning the whole amount at once. This happens as the credits-per-token ratio drops below 1e18.
Severity: Low.
Mitigation: Acknowledged. Amounts are extremely small.

Issue: mint operation is not additive.
Rule(s) broken: mint is additive.
Description: Minting in parts leads to smaller balance in the end then minting the whole amount at once.
Severity: Low.
Mitigation: As it is the intended side of the tradeoff, no action is taken.

Issue: Break redemption into multiple calls to avoid fees.
Description: Since redemption fees go to the communal funds pool, breaking the redemption into multiple calls (and rebasing the supply between them) can allow the user to gain back some of the fees he paid. The percentage of of fees that the user is able to get back (or avoid paying) is bounded from above by his share of the OUSD total supply.
Severity: Medium.
Mitigation: Acknowledged.

Issue: Break mint into multiple calls to avoid rebasing.
Description: If a large rebasing is waiting to occur, a user can mint a large sum of OUSD in small portions to avoid the rebasing threshold until the very end of the mint, thus gaining a large portion of the rebasing revenues.
Severity: Medium.
Mitigation: Acknowledged.

Issue: Loss of fund value for non-rebasing accounts due to unstable stablecoin
Description: When the value of one of the supported stablecoins falls below $1, a malicious user can mint large amount of OUSD tokens using this stablecoin, rebase the total supply, and then redeem all his OUSD tokens to get a proportion of higher valued stablecoins at the expense of the non-rebasing accounts in the system.
Severity: Low.
Mitigation: Acknowledged. OUSD risks document states that the ratio of stablecoins is not guaranteed in the event of a stablecoin peg loss.

Issue: Redemption of small amounts avoids fees.
Description: Assuming a fee of 0.5%, the redemption fee for amounts smaller than 2e-16 will be 0.
Severity: Low.
Mitigation: Gas prices render such a scenario impossible.

Issue: Burn of 0 can update the amount of tokens.
Rule(s) broken: burn is neutral for 0 tokens.
Description: Due to dusting-off of credits, if the user has 1 credit and asks to burn 0 tokens, they will have 0 credits and thus 0 tokens. This is counterintuitive to have an amount of 0 affecting the state.
Severity: Low.
Mitigation: Fixed.

Issue: mint and burn use a return command but are not returning anything
Description: The internal functions _mint and _burn do not return a value, yet mint and burn return the “values” returned from those.
Severity: Very low.
Mitigation: Fixed.

Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and any cases not covered by the specification are not checked by the Certora Prover.

We hope that this information is useful, but provide no warranty of any kind, express or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Notations

1. ✔ indicates the rule is formally verified on the latest commit. We write ✔* when the rule was verified on a simplified version of the code (or under some assumptions).
2. ❌ indicates the rule is violated in the version tested.
3. ✍ indicates the rule is not yet formally specified.
4. 🔁 indicates the rule is postponed **(<due to other issue, low priority?>)** .
5. We use Hoare triples of the form {p} C {q}, which means that if the execution of program C starts in any state satisfying p, it will end in a state satisfying q. In Solidity, p is similar to require, and q is similar to assert.
   The syntax {p} ($C_1 \sim C_2$) {q} is a generalization of Hoare rules, called relational properties. {p} is a requirement on the states before C_1 and C_2, and {q} describes the states after their executions. Notice that C_1 and C_2 result in different states. As a special case, $C_1 \sim$op C_2, where op is a getter, indicating that C_1 and C_2 result in states with the same value for op.

Verification of OUSD Token

The **OUSD token** is an ERC20 token that is potentially gaining value thanks to investments made using its backing stablecoins. OUSD accounts could be either *"rebasing"* or *"non-rebasing"* depending if their amount of OUSD is dynamically updating as the system is synchronized with the underlying investments. This synchronization process is an externally executed operation we refer to as *"rebasing"* in this document.

The actual stored value is denominated in *credits*. How much these stored credits are worth in OUSD depends on a ratio *creditsPerToken*. The *creditsPerToken* ratio is fixed for non-rebasing accounts, and variable in rebasing accounts. *creditsPerToken* ratio is given as a mantissa, where 1 is equal 1e18. For example, if the credits per token ratio is 2e18, then every 2 units of credits are worth one OUSD token. Another example is that if credits per token ratio is 5e17, then every 1 unit of credit is worth 2 OUSD tokens.

As a convenience, we will use the following notations for the different units, and mark **uint** amounts accordingly.

- **OT** for OUSD tokens
- **C** for credits
- **C/OT** for credits per token ratio

All the amounts are scaled by 1e18.

Functions

OUSD.totalSupply() : uint (OT)
(ERC20 standard.) Returns the total amount of OUSD tokens in circulation.

OUSD.balanceOf(address a) : uint (OT)
(ERC20 standard.) Returns the amount of OUSD tokens held by an account *a*.

OUSD.allowance(address owner, address spender) : uint (OT)
(ERC20 standard.) Returns the amount of OUSD tokens that an account *owner* allows an account *spender* to spend on *owner*'s behalf.

OUSD.transfer(address from, address to, uint (OT) amount) : bool
(ERC20 standard.) Transfers *amount* OUSD tokens from account *from* to account *to*.

OUSD.mint(address minter, uint (OT) amount) : bool
(ERC20 extended standard.) Increases supply by *amount* OUSD tokens and transfers them to the account *minter*.

OUSD.burn(address burned, uint (OT) amount) : bool
(ERC20 extended standard.) Decreases supply by *amount* OUSD tokens and transfers them from the account *burned*.

OUSD.rebasingCreditsPerToken() : uint (C/OT)
Returns the credits per token ratio for rebasing accounts.

OUSD.nonRebasingCreditsPerToken(address a) : uint (C/OT)
Returns the credits per token ratio for a non-rebasing account *a*.

OUSD.nonRebasingSupply() : uint (OT)
Returns the amount of OUSD tokens in circulation that are in the ownership of non-rebasing accounts.

OUSD.rebasingCredits() : uint (C)
Returns the amount of credits that are in the ownership of rebasing accounts.

OUSD.rebaseState(address a) : uint8
Returns whether the account *a* has set to opt-in for rebasing, opt-out for rebasing, or no explicit choice. If no explicit choice was made, the system defines a default behavior where externally-owned accounts are rebasing, and contracts are non-rebasing. (See **isNonRebasingAccount**)

OUSD.isNonRebasingAccount(address a) : bool
Returns whether the account *a* is a rebasing or non-rebasing account.

OUSD.maxSupply() : uint (OT)
Returns the maximal supply of OUSD tokens allowed in circulation.

OUSD.rebaseOptIn(address a)
Sets account *a* as a rebasing account.

OUSD.rebaseOptOut(address a)
Sets account *a* as a non-rebasing account.

Vault.totalValue(): uint (OT)
The total number of stablecoins stored in the vault and in strategies.

Properties

1. **Total supply of OUSD must be below max supply of OUSD ❌**

   ```
   OUSD.totalSupply() ≤ OUSD.maxSupply()
   ```

2. **The balanceOf function should never revert ✔**

3. **The rebasing credits-per-token ratio must be positive ✔**

   ```
   OUSD.rebasingCreditsPerToken() > 0
   ```

4. **Sender cannot decrease other users' balances ✍**
   If the operation's sender does not have an allowance for an account *a*, then it should be impossible for the sender to execute an operation that reduces *a*'s balance.
   [Status: burn() violates the rule. It should be proved that the Vault can only call OUSD's burn() for the msg.sender]

   ```
   { sender != a ∧
     OUSD.allowance(a, sender) = 0 ∧
     b = OUSD.balanceOf(a) }
         OUSD.op()
   { balanceOf(a) ≥ b }
   ```

5. **Only rebaseOptIn() & rebaseOptOut() may modify the rebase state ✔**
   For an account *a*, the only way to change its *rebaseState* is by invoking either rebaseOptIn() or rebaseOptOut().

   ```
   { state = OUSD.rebaseState(a) }
         OUSD.op()
   { OUSD.rebaseState(a) = state },
   ```
   where `OUSD.op ∉ { OUSD.rebaseOptIn, OUSD.rebaseOptOut }`

6. **Total supply of OUSD is consistent ❌**

```
OUSD.totalSupply()
      = OUSD.rebasingCredits() / OUSD.rebasingCreditsPerToken()
            + OUSD.nonRebasingSupply()
```

7. **The basic preconditions for transfer are checked ✔**
   According to the ERC20 standard, a transfer to a non-zero address of non-zero amount should succeed only if the sender's balance is at least amount.

```
{ precond = OUSD.balanceOf(from) >= value ∧
  to != 0 ∧
  amount != 0 }
      successful = OUSD.transfer(from, to, amount)
{ successful }
```

8. **Transfer updates balances as expected ✍**
   According to the ERC20 standard, a transfer to a non-zero address of non-zero amount should deduct the amount from the sender's balance and add the amount to the recipient's balance.

```
{ b_from = OUSD.balanceOf(from) ∧
  b_to = OUSD.balanceOf(to) ∧
  to != 0 ∧
  value != 0 }
      successful = OUSD.transfer(from, to, amount)
{ ((from != to ∧ successful) ⇒
      OUSD.balanceOf(from) = b_from - amount ∧
      OUSD.balanceOf(to) = b_to + amount)
  ∧
  ((from == to ∨ ¬successful) ⇒
      OUSD.balanceOf(from) = b_from ∧
      OUSD.balanceOf(to) = b_to)
}
```

9. **Solvency** ✍
   The backing amount of stablecoins stored in the vault is covering the available supply of OUSD.

   ```
   vault.totalValue() ≥ OUSD.totalSupply()
   ```

10. **Monotonicity of rebasing credits per token** ✍
    The rebasing credits per token is monotonically decreasing.
    (Opting-out should only hurt)

    ```
    { rcpt = rebasingCreditsPerToken() }
       op()
    { rebasingCreditsPerToken() ≤ rcpt }
    ```

11. **Burn is additive** ❌
    Burning OUSD tokens from an account *burned* is additive, i.e., it can be performed either all at once or in steps.

    ```
    OUSD.burn(burned,x); OUSD.burn(burned, y); ~ OUSD.burn(burned, x+y)
    { OUSD.balanceOf1(burned) = OUSD.balanceOf2(burned) }
    ```

12. **Mint is additive** ❌
    Minting OUSD tokens to an account *minter* is additive, i.e., it can be performed either all at once or in steps.

    ```
    OUSD.mint(minter,x); OUSD.mint(minter, y); ~ OUSD.mint(minter, x+y)
    { OUSD.balanceOf1(minter) = OUSD.balanceOf2(minter) }
    ```

13. **Burn is neutral for 0 tokens [By Origin]** ❌
    Burning 0 OUSD tokens should have no effect on the balance of OUSD tokens of the user.

    ```
    { b = OUSD.balanceOf(burned) }
          OUSD.burn(burned,0)
    { b = OUSD.balanceOf(burned) }
    ```