Origin Dollar

Security Assessment

December 21, 2020

Prepared For:
Josh Fraser | *Origin Protocol*
josh@originprotocol.com

Prepared By:
Alexander Remie | *Trail of Bits*
alexander.remie@trailofbits.com

Dominik Teiml | *Trail of Bits*
dominik.teiml@trailofbits.com

Changelog:

November 18, 2020:	Draft final report delivered
December 18, 2020:	Added Appendix E with retest results
December 21, 2020:	Updated Appendix E to reflect newest changes

Executive Summary

From November 2 through November 17, 2020, Origin Protocol engaged Trail of Bits to review the security of Origin Dollar. Trail of Bits conducted this assessment over the course of 4 person-weeks with 2 engineers working from 81431fd.

The first week, we gained an overall understanding of the codebase. We reviewed the OUSD contract and started reviewing the Vault contracts. Our focus was on the rebasing process and invariants of OUSD, the allocation of funds inside VaultCore, and the VaultAdmin contract. In week two, we focused on the AaveStrategy and CompoundStrategy contracts, and the various Oracle and Governance-related contracts. In the two days of the final week, we dedicated further review to the VaultCore contract.

Our review resulted in 23 findings ranging from high to informational severity. One of the high-severity issues is of low difficulty and would allow an attacker to drain the funds of the system. Several other high-severity issues were of higher difficulty and originated in the governance-related contracts and OUSD contract. The high-severity issues we found are:

- Missing input validation when depositing stablecoins for OUSD, allowing an attacker to drain the funds of the contract. (TOB-OUSD-010).
- Incorrect access controls prohibiting Governance Proposals from being canceled (TOB-OUSD-006).
- Missing check that could block the retrieval of OUSD account balances (TOB-OUSD-002).
- Lack of access controls allowing Governance Proposal transactions to be executed separately instead of atomically (TOB-OUSD-008).
- Lack of input validation could allow Governance admin role takeover (TOB-OUSD-009).
- External calls in a loop could lead to DoS (TOB-OUSD-020).
- Not checking the return value could lead to a user not getting back collateral when redeeming their OUSD (TOB-OUSD-019).
- OUSD allows transferring more tokens than a user has due to rounding issues (TOB-OUSD-022).
- OUSD violates a common ERC20 invariant (TOB-OUSD-023).

We also found several issues related to input validation (TOB-OUSD-010, TOB-OUSD-001, TOB-OUSD-013). There were also several best practices that were not adhered to: not using SafeMath (TOB-OUSD-003), unsafe last array element deletion (TOB-OUSD-012), not checking ERC20 transfer return value (TOB-OUSD-004), missing events for important operations (TOB-OUSD-021), variable shadowing (TOB-OUSD-016) , and missing return statements (TOB-OUSD-017). Additional code quality points can be found in Appendix C.

Overall, the Origin Dollar contracts are not yet ready for deployment. The high severity issue that allowed contract funds to be drained, caused by missing input validation and not taking reentrancy into account, exemplifies the current state of the project. Missing input validation in dozens of functions and issues in Governance contracts further indicates that more work is required before deployment. Finally, several issues were detected using automated analysis with Slither and crytic.io, including a high severity vulnerability, highlighting the processes for testing and verification that need improvement.

Trail of Bits recommends addressing the short- and long-term findings presented in this report. We also recommend a feature freeze until the existing features are properly documented and their assumptions tested in-depth. Finally, due to the prevalence of high-severity findings, we recommend additional focused security reviews once the reported findings have been addressed.

Update December 21, 2020: Trail of Bits reviewed fixes provided by Origin Protocol for the issues described in this report. Further information can be found in Appendix F. Fix Log.

Project Dashboard

Application Summary

Name	Origin Dollar
Version	81431fd
Type	Solidity
Platforms	Ethereum

Engagement Summary

Dates	November 2 - November 17, 2020
Method	Whitebox
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	8	■■■■■■■
Total Medium-Severity Issues	1	■
Total Low-Severity Issues	6	■■■■■■
Total Informational-Severity Issues	5	■■■■■
Total Undetermined-Severity Issues	3	■■■
Total	23	

Category Breakdown

Data Validation	9	■■■■■■■■
Undefined Behavior	8	■■■■■■■■
Access Controls	1	■
Arithmetic	1	■
Standards	1	■
Timing	1	■
Auditing and Logging	1	■
Denial of Service	1	■

Total	23	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Weak.** We found many issues with privileged roles in the system, e.g. TOB-OUSD-006 and TOB-OUSD-008.
Arithmetic	**Weak.** Use of SafeMath is inconsistent and untrusted data is not always validated before being accepted (TOB-OUSD-002 , TOB-OUSD-003). We also discovered several issues due to rounding errors (TOB-OUSD-022).
Assembly Use	**Moderate.** Assembly use is sparse, however, it is used in a way not conforming to a standard (see TOB-OUSD-005).
Decentralization	**Weak.** The governor guardian and other privileged roles hold substantial power over the system, including the ability to set system-wide parameters and upgrade implementations.
Upgradeability	**Moderate.** The system partially complies with EIP-1967. See TOB-OUSD-005.
Function Composition	**Moderate.** The code is divided into folders with contracts grouped according to their functionality. The use of Solidity inheritance and libraries correctly separates different layers of abstraction. However, the lack of extensive documentation and careful testing makes the code more difficult to review than expected. Some contracts inherit contracts that are not used (TOB-OUSD-007).
Front-Running	**Further Investigation Required.**
Key Management	**Not Considered.**
Monitoring	**Weak.** We found that events to monitor the contracts were missing or confusing (see TOB-OUSD-013, TOB-OSUD-021). Additionally, there is no documented incident response plan.
Specification	**Moderate**. The code contains minimal documentation. There is a high-level description of the system, but there is no detailed (formal or semi-formal) specification of every contract.
Testing & Verification	**Weak.** We found issues like reentrancy attacks (TOB-OUSD-010), highlighting the lack of comprehensive use of tools like Slither or Echidna.

Engagement Goals

The engagement was scoped to provide a security assessment of OUSD smart contracts in the `origin-dollar` repository.

Specifically, we sought to answer the following questions:

- Are appropriate access controls set for the user/controller roles?
- Does arithmetic regarding token and vault operations hold?
- Does the governance work as expected?
- Can participants manipulate or block token, vault, or governance operations?
- Can participants steal or lose tokens?
- Does the rebasing of the token work correctly?
- Can participants perform denial-of-service or phishing attacks against any of the system components?

Coverage

The engagement was focused on the following components:

- **Vault:** The vault contracts are the entry into the system and allow users to deposit stablecoins and get OUSD in exchange or redeem OUSD to get back stablecoins. We manually reviewed this contract and used automatic tools to identify reentrancy bugs and reviewed the other properties of the Vault.
- **Strategies:** The strategy contracts are used by the vault to deposit and withdraw the stablecoins to earn interest. We reviewed the correctness of the Aave and Compound Strategy contracts.
- **Oracles:** The OUSD project makes use of Uniswap and Chainlink oracles to retrieve various prices. We reviewed the correct conversion between the different decimals used and the correct functioning of the price calculation functions, as well as other properties of the oracle contracts.
- **Governance:** The governance contracts allow anyone to propose changes to contract parameters that, when accepted by the guardian, can be executed after a time delay has passed. We reviewed these contracts to ensure the governance process cannot be subverted and that governance functionality works as expected.
- **Origin Dollar token:** The vault mints or burns tokens every time a user deposits/redeems collateral stablecoins. This contract implements a standard ERC20 token. We verified that all the expected properties are correctly implemented.
- **Access controls.** Many parts of the system expose privileged functionality, such as upgradability functions. We reviewed these functions to ensure they can only be

triggered by the intended actors and that they do not contain unnecessary privileges that may be abused.
- **Arithmetic.** We reviewed calculations for logical consistency, rounding issues, and scenarios where reverts due to overflow may negatively impact the use of the token.

We did not review the following contracts:
- CurveStrategy

Off-chain code components were outside the scope of this assessment.

Automated Testing and Verification

Trail of Bits used automated testing techniques to enhance coverage of certain areas of the contracts. Automated testing techniques augment our manual security review but do not replace it. For this audit, we employed Echidna, a smart contract fuzzer. This tool can rapidly test security properties via malicious, coverage-guided test case generation.

We used Echidna to verify that the OUSD token follows standard ERC20 properties. These properties were generated using slither-prop, our open-source tool that collects common smart contract properties to test.

The following table details the high-level description of every tested property and the outcome after running it for at least 50,000 iterations.

#	Property	Result
1	Transferring tokens to the null address (`0x0`) causes a revert.	PASSED
2	The null address (`0x0`) owns no tokens.	**FAILED** (TOB-OUSD-002)
3	Transferring a valid amount of tokens to a non-null address reduces the current balance.	**FAILED** (TOB-OUSD-002)
4	Transferring an invalid amount of tokens to a non-null address reverts or returns false.	**FAILED** (TOB-OUSD-022)
5	Self-transferring a valid amount of tokens keeps the current balance constant.	**FAILED** (TOB-OUSD-002)
6	Approving overwrites the previous allowance value.	PASSED
7	The balances are consistent with the `totalSupply`.	**FAILED** (TOB-OUSD-002, TOB-OUSD-023)

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

☐ **Add validation to the `setVaultBuffer` function to disallow a value above 1e18.** If the value exceeds 1e18 an underflow might happen under specific circumstances. TOB-OUSD-001

☐ **Add validation to ensure that `rebasingCreditsPerToken` is always non-zero.** If this value becomes zero the balances of accounts cannot be queried anymore due to a division by zero. TOB-OUSD-002

☐ **Use SafeMath for all mathematical operations unless otherwise desired.** Not using SafeMath increases the risk of under-/overflows. TOB-OUSD-003

☐ **As in other places throughout the contracts, also use `safeTransfer` here.** Not using safeTransfer leads to unexpected results depending on the token used. TOB-OUSD-004

☐ **Make sure the contracts comply with EIP-1967 if that is the goal.** Currently the contracts are only half compliant with EIP-1967. TOB-OUSD-005

☐ **Add a function to the `Governor` that calls `Timelock.cancelTransaction`.** Currently it's not possible for cancelTransaction to be called by anyone, which is a valuable feature to have. TOB-OUSD-006

☐ **Remove the `InitializableGovernable` contract from the project and let all three oracle contracts directly inherit from `Governable` instead.** The oracle contracts are not upgradeable and thus they do not make use of the Initializable-related code. TOB-OUSD-007

☐ **Only allow the `admin` to call `Timelock.executeTransaction`.** By allowing anybody to execute transactions separately, the possible atomicity requirement of transactions bundled in a Proposal can be broken. TOB-OUSD-008

☐ **Add a check that prevents `setPendingAdmin` to be included in a `Proposal`.** There already exist specific functions to update the pendingAdmin. TOB-OUSD-009

☐ **Add checks that cause `mintMultiple` to revert if the amount is zero or the asset is not supported. Add a reentrancy guard to the `mint`, `mintMultiple`, `redeem`, and `redeemAll` functions.** These critical bugs should be prevented as they allow draining the contract. TOB-OUSD-010

☐ **Update the checks to allow the `minDrift` and `maxDrift` values and add unit tests to ensure the boundary values are allowed**. Boundary values should be handled correctly. TOB-OUSD-011

☐ **Remove the last element of an array using `pop()`.** Making use of Solidity-provided security mitigations should be used unless there is a good reason not to. TOB-OUSD-012

☐ **Add a check that causes a revert if the strategy does not exist.** Allowing a non-existent strategy's weight to be set shouldn't be allowed. TOB-OUSD-013

☐ **Add a minimum expected redeem value argument to the redeem functions.** This protects users from sudden changes in the redeem fee. TOB-OUSD-014

☐ **Transfer `amountWithdrawn` instead of the entire contract's ERC20 balance.** Having the recipient receive more tokens than withdrawn is incorrect. TOB-OUSD-015

☐ **Remove the shadowed variables (`_allowances` and `_totalSupply`) in `OUSD`.** The shadowed variables can lead to unexpected behavior when used. TOB-OUSD-016

☐ **Add the missing return statement(s) or remove the return type in `VaultCore.rebase()` and `VaultCore.rebase(bool)`. Properly adjust the documentation as necessary.** The lack of return statement leads to these functions always returning zero, which can lead to unexpected behavior for the caller. TOB-OUSD-017

☐ **Ensure `ChainlinkOracle and OpenUniswapOracle` inherit from `IPriceOracle`, and that `RebaseHooks` inherits from `IRebaseHooks`.** The lack of inheritance can lead the implementation to be out of synchronization with their interface. TOB-OUSD-018

☐ **Check the return value for all the calls mentioned in TOB-OUSD-019.** The lack of return value checks is error prone and can lead to unexpected behaviors.

☐ **Review all the loops mentioned in TOB-OUSD-020 and either allow iteration over part of the loop, or to remove elements.** Having a loop that reverts could block certain functions from being executed and cause DoS.

☐ **Add events for all critical operations listed in TOB-OUSD-021.** Events help to monitor the contracts and detect suspicious behavior.

☐ **Make sure the balance is correctly checked before performing all the arithmetic operations.** This will make sure it does not allow to transfer more than expected. TOB-OUSD-022

☐ **Clearly indicate all common invariant violations for users and other stakeholders.** In some cases, you will be forced to operate in a different manner than may be expected by users. Make sure these conditions are clearly discoverable. TOB-OUSD-023

Long term

 Validate the function inputs in all the contracts/libraries. Validating all inputs to a function allows returning descriptive error messages to the caller. Instead of the function reverting later on because of some invalid value and returning a nondescript revert message. TOB-OUSD-001

 Write a specification of each function and check it through fuzzing or verify it with symbolic execution. The system relies on invariants that must hold to ensure its security. Several issues would have been avoided with a proper testing or verification.

- Check for arithmetic invariants TOB-OUSD-001, TOB-OUSD-002, TOB-OUSD-003, TOB-OUSD-011
- Check the proper access controls TOB-OUSD-008, TOB-OUSD-013,
- Check that ERC20 transfers are transferring the expected amount TOB-OUSD-015, TOB-OUSD-022

 Subscribe to Crytic. Crytic catches many of the bugs reported. TOB-OUSD-004, TOB-OUSD-010, TOB-OUSD-016, TOB-OUSD-017, TOB-OUSD-018, TOB-OUSD-019, TOB-OUSD-020

 Implement EIP's in their entirety if the goal is to be compliant. To be able to tell users that you're fully adhering to EIP's. TOB-OUSD-005

 Consider letting `Governor` inherit from `Timelock`. This would allow a lot of functions and code to be removed and significantly lower the complexity of these two contracts. TOB-OUSD-006, TOB-OUSD-009

 Get rid of all unused code in the codebase. This adds additional risk and increases the attack surface, without any gains as the code is not used. TOB-OUSD-007

 Identify other places in the codebase with boundary checks and ensure that they work as expected by writing unit tests. Having unit tests that ensure boundary values are correctly allowed or disallowed prevents these types of bugs. TOB-OUSD-011

 Keep track and make use of new Solidity features that prevent common bugs. The Solidity language is constantly improving. Occasionally a language improvement directly addresses a source of common bugs. By making use of these improvements the bug can be prevented, while at the same time the code quality increases (no need for custom checks). TOB-OUSD-012

 Identify functions that might be affected by a sudden contract parameter change and add mitigations to protect users from such surprises. Protecting users from sudden changes that could negatively affect them prevents them from being surprised and disappointed. TOB-OUSD-014

☐ **Consider using a blockchain monitoring system to track suspicious behavior in the contracts.** The system relies on the correct behavior of several contracts. A monitoring system which tracks critical events would allow quick detection of any compromised system components. TOB-OUSD-021

☐ **Design the system to preserve as many commonplace invariants as possible.** That will allow users and third-party contracts to interact with the OUSD token without unexpected consequences. TOB-OUSD-023

Findings Summary

#	Title	Type	Severity
1	Invalid vaultBuffer could revert allocate	Data Validation	Low
2	OUSD.changeSupply should require rebasingCreditsPerToken > 0	Data Validation	High
3	SafeMath is recommended in OUSD._executeTransfer	Data Validation	Informational
4	Transfers could silently fail without safeTransfer	Undefined Behavior	Informational
5	Proxies are only partially EIP-1967-compliant	Standards	Informational
6	Queued transactions cannot be cancelled	Access Controls	High
7	Unused code could cause problems in future	Undefined Behavior	Undetermined
8	Proposal transactions can be executed separately and block Proposal.execute call	Undefined Behavior	High
9	Proposals could allow Timelock admin takeover	Data Validation	High
10	Reentrancy and untrusted contract call in mintMultiple	Data Validation	High
11	Off-by-one minDrift/maxDrift causes unexpected revert	Data Validation	Low
12	Unsafe last array element removal poses future risk	Arithmetic	Undetermined
13	Strategy targetWeight can be set for non-existent strategy	Data Validation	Low
14	Lack of minimal redeem value might lead to less return than expected	Timing	Medium
15	withdraw allows redeemer to withdraw accidentally sent tokens	Undefined Behavior	Low

16	Variable shadowing from OUSD to ERC20	Undefined Behavior	Low
17	VaultCore.rebase functions have no return statements	Undefined Behavior	Low
18	Multiple contracts are missing inheritances	Undefined Behavior	Informational
19	Lack of return value checks can lead to unexpected results	Undefined Behavior	Undetermined
20	External calls in loop can lead to denial of service	Denial of Service	High
21	No events for critical operations	Auditing and Logging	Informational
22	OUSD allows users to transfer more tokens than expected	Data Validation	High
23	OUSD._totalSupply can be arbitrary, even smaller than user balances	Data Validation	High

1. Invalid vaultBuffer could revert allocate

Severity: Low
Type: Data Validation
Target: `VaultAdmin.sol, VaultCore.sol`

Difficulty: High
Finding ID: TOB-OUSD-001

Description

The lack of input validation when updating the `vaultBuffer` could cause token allocations inside `allocate` to revert when no revert is expected.

```
    function setVaultBuffer(uint256 _vaultBuffer) external onlyGovernor {
        vaultBuffer = _vaultBuffer;
    }
```

Figure 1.1: `VaultAdmin.sol#L50-L52`

Every account can call `allocate` to allocate excess tokens in the `Vault` to the strategies to earn interest.

The `vaultBuffer` indicates how much percent of the tokens inside the `Vault` to allocate to strategies (to earn interest) when `allocate` is called. The `setVaultBuffer` function allows `vaultBuffer` to be set to a value above 1e18(=100%). This function can only be called by the `Governor` contract, which is a multi-sig. Mistakenly proposing 1e19(=1000%) instead of 1e18 might not be noticed by the `Governor` participants.

If the `vaultBuffer` is above 1e18 and at least one of the strategies has been allocated some tokens, the function will simply return. However, in case none of the strategies have yet been allocated any tokens, the `vaultBuffer` is subtracted from 1e18 causing an underflow. Depending on the result of the underflow, this could cause a revert when the `Vault` contract tries to transfer tokens to a strategy since the contract does not possess that amount of tokens. What would be expected in this situation is for no allocations to occur and the transaction to successfully execute, instead of reverting.

This issue could be mitigated by preventing the underflow by e.g. using SafeMath. However, the root cause is the lack of input validation in `VaultAdmin`. Such is the case for most of the other functions inside `VaultAdmin`.

This issue serves as an example as there is no input validation in any function protected by the `onlyGovernor` modifier.

Exploit Scenario

No strategies have been allocated any tokens yet. Bob intends to create a proposal to update the `vaultBuffer` to 100%, but instead of 1e18 mistakenly passes in 1e19. None of the other participants in `Governor` notice this mistake and the proposal is approved. The `vaultBuffer` is updated to 1e19 and suddenly calls to `allocate` cause a revert instead of successful execution.

Recommendation
Short term, add validation to the `setVaultBuffer` function to disallow a value above `1e18`.

Long term, validate the function inputs in all the contracts/libraries. Add input validation to all functions callable by the Governor. Consider using SafeMath for all arithmetic or proving no under-/overflows can happen through [Manticore](https://github.com/trailofbits/manticore).

2. OUSD.changeSupply should require rebasingCreditsPerToken > 0

Severity: High
Type: Data Validation
Target: `OUSD.sol`

Difficulty: High
Finding ID: TOB-OUSD-002

Description

In `OUSD.sol`, `changeSupply` is used to inflate or deflate the money supply of rebasing accounts.

```
function changeSupply(uint256 _newTotalSupply)
        external
        onlyVault
        returns (uint256)
    {
        require(_totalSupply > 0, "Cannot increase 0 supply");

        ...


        _totalSupply = _newTotalSupply;

        ...

        rebasingCreditsPerToken = rebasingCredits.divPrecisely(
            _totalSupply.sub(nonRebasingSupply)
        );
```

Figure 2.1: `OUSD.sol#L477-L499`

In particular, for any reasonable values for `rebasingCredits` and `nonRebasingSupply`, it is possible to set a `_newTotalSupply` so `rebasingCreditsPerToken = 0`. This would break a lot of invariants in the contract, e.g. `balanceOf` will be reverting for rebasing accounts.

Exploit Scenario

An external contract checks for the `OUSD` balance of an account. It is expecting the call to succeed, but instead, it reverts, leading to unintended consequences.

Recommendation

Short term, add validation to ensure that `rebasingCreditsPerToken` is always non-zero.

Long term, use Echidna to ensure that all invariants always hold.

3. SafeMath is recommended in `OUSD._executeTransfer`

Severity: Informational
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-OUSD-003
Target: `OUSD.sol`

Description

`_executeTransfer`, after exchanging the corresponding amount of credits, updates the accounting state variables:

```
        } else if (isNonRebasingTo && isNonRebasingFrom) {
            // Transfer between two non rebasing accounts. They may have
            // different exchange rates so update the count of non rebasing
            // credits with the difference
            nonRebasingCredits =
                nonRebasingCredits +
                creditsCredited -
                creditsDeducted;
        }
```

Figure 3.1: `OUSD.sol#L187-L195`

While it can be shown if the from address is non-rebasing, than `nonRebasingCredits >= creditsDeducted`, we nevertheless recommend using SafeMath unless there is a good reason not to. Reverting in this case would likely be safe failure, while an underflow might be a catastrophic one.

Recommendation

Short term, use SafeMath for all mathematical operations unless otherwise desired.

Long term, consider using [Manticore](https://github.com/trailofbits/manticore) or [Echidna](https://github.com/crytic/echidna) to check for under/overflow issues.

4. Transfers could silently fail without safeTransfer

Severity: Informational
Difficulty: High
Type: Undefined Behavior
Finding ID: TOB-OUSD-004
Target: `VaultAdmin.sol, InitializableAbstractStrategy.sol`

Description
Several functions do not check the `ERC20.transfer` return value. Without a return value check, the transfer is error-prone, which may lead to unexpected results.

```
function transferToken(address _asset, uint256 _amount)
    external
    onlyGovernor
{
    IERC20(_asset).transfer(governor(), _amount);
}
```

Figure 4.1: `VaultAdmin.sol#L235-L240`

`VaultAdmin.transferToken` calls `ERC20.transfer` without checking the return value. As a result, the governor withdrawing ERC20 tokens might fail while it appears to succeed.

The following areas were identified as missing the appropriate checks on return values:

- `InitializableAbstractStrategy.transferToken`
- `InitializableAbstractStrategy.collectRewardToken`
- `VaultAdmin.transferToken`

The `collectRewardToken` function transfers the `rewardToken` of the strategy. The `rewardToken` of every strategy seems to throw on failure, and no return value check should be necessary. Still, for some reason this might change in the future, although very unlikely. Thinking in terms of defense in depth would also use `safeTransfer` here.

Exploit Scenario
Bob accidentally transfers 100 BAT to the `VaultAdmin` contract. The governor of the `VaultAdmin` wants to withdraw these tokens by calling `transferToken`. However, due to a mistake he enters 1000 instead of 100 as the `_amount`. The transaction succeeds making the governor believe he withdrew the 100 BAT, while it actually silently failed.

Recommendation
Short term, as in other places throughout the contracts, also use `safeTransfer` here.

Long term, subscribe to [Crytic](). Crytic catches this bug class automatically.

5. Proxies are only partially EIP-1967-compliant

Severity: Informational
Type: Standards
Target: `InitializeGovernedUpgradeabilityProxy.sol`

Difficulty: N/A
Finding ID: TOB-OUSD-005

Description
The `InitializeGovernedUpgradeabilityProxy` saves the implementation in a storage slot fully compliant with [EIP-1967](EIP-1967).

```
    assert(
        IMPLEMENTATION_SLOT ==
            bytes32(uint256(keccak256("eip1967.proxy.implementation")) - 1)
    );
```

Figure 5.1: `InitializeGovernedUpgradeabilityProxy.sol#L32-L35`

For the `admin` however, the contract calls `Governable`'s `_setGovernor`, which saves it at a different storage slot than that of the EIP.

```
    // Storage position of the owner and pendingOwner of the contract
    bytes32 private constant governorPosition =
0x7bea13895fa79d2831e0a9e28edede30099005a50d652d8957cf8a607ee6ca4a;
    //keccak256("OUSD.governor");
```

Figure 5.2: `Governable.sol#L11-L14`

Exploit Scenario
A third party could make the educated assumption that the contract conforms to EIP-1967. They expect the governor to be stored at `keccak256("eip1967.proxy.admin") - 1`, but instead it is not. That could lead to unforeseen consequences.

Recommendation
Short term, make sure the contracts comply with EIP-1967 if that is the goal.

Long term, implement EIP's in their entirety if the goal is to be compliant.

6. Queued transactions cannot be canceled

Severity: High
Type: Access Controls
Target: `Governor.sol, Timelock.sol`

Difficulty: Medium
Finding ID: TOB-OUSD-006

Description

The `Governor` contract contains special functions to set it as the admin of the `Timelock`. Only the `admin` can call `Timelock.cancelTransaction`. There are no functions in `Governor` that call Timelock.cancelTransaction. This makes it impossible for `Timelock.cancelTransaction` to ever be called.

```
function __acceptAdmin() public {
    require(
        msg.sender == guardian,
        "Governor::__acceptAdmin: sender must be gov guardian"
    );
    timelock.acceptAdmin();
}
```

Figure 6.1: `Governor.sol#L206-L212`

The `Governor` becomes the `admin` of `Timelock`.

```
function cancelTransaction(
    address target,
    uint256 value,
    string memory signature,
    bytes memory data,
    uint256 eta
) public {
    require(
        msg.sender == admin,
        "Timelock::cancelTransaction: Call must come from admin."
    );
```

Figure 6.2: `Timelock.sol#L140-L150`

The `cancelTransaction` function can only be called by the `admin`.

If Origin Protocol is made aware of an incorrect transaction but is unable to cancel it, the next best thing to do might be to quickly change the `admin`. However, due to the `delay` requirement of queued transactions, the `admin` change transaction would become executable only after the transaction which should be canceled.

Exploit Scenario

Bob creates a proposal with five transactions. One of the transactions contains an incorrect function argument. The `guardian` doesn't notice this at first and queues the `Proposal`. Somebody notices this and notifies Origin Protocol about the incorrect transaction. Origin Protocol wants to cancel that specific transaction but finds out that it's not possible to call `Timelock.cancelTransaction`.

Recommendation

Short term, add a function to the `Governor` that calls `Timelock.cancelTransaction`. It is unclear who should be able to call it, and what other restrictions there should be around cancelling a transaction.

Long term, consider letting `Governor` inherit from `Timelock`. This would allow a lot of functions and code to be removed and significantly lower the complexity of these two contracts.

7. Unused code could cause problems in future

Severity: Undetermined | Difficulty: High
Type: Undefined Behavior | Finding ID: TOB-OUSD-007
Target: `ChainlinkOracle.sol, MixOracle.sol, OpenUniswapOracle.sol`

Description

The three oracle contracts are not upgradeable, yet contain code meant for upgradeable contracts. This unnecessarily increases the attack surface and could cause problems in the future if any of this unused code causes a low-level bug.

```
contract InitializableGovernable is Governable, Initializable {
    function _initialize(address _governor) internal {
        _changeGovernor(_governor);
    }
}
```

Figure 7.1: `InitializableGovernable.sol#L13-L17`

All three oracle contracts inherit from the above contract. Since none of the oracle contracts is upgradeable the above function is never called. Also, the `Initializable` contract is included but never used in the oracle contracts.

Exploit Scenario

A low-level bug is discovered which affects contracts that have a storage layout specifically as created by the `Initializable` contract. Even though not used by the oracle contracts, the bug affects them.

Recommendation

Short term, remove the `InitializableGovernable` contract from the project and let all three oracle contracts directly inherit from `Governable` instead.

Long term, get rid of all unused code in the codebase.

8. Proposal transactions can be executed separately and block Proposal.execute call

Severity: High
Type: Undefined Behavior
Target: `Governor.sol, Timelock.sol`

Difficulty: Low
Finding ID: TOB-OUSD-008

Description

Missing access controls in the `Timelock.executeTransaction` function allow `Proposal` transactions to be executed separately. Circumventing the `Governor.execute` function. This means that even though the `Proposal.executed` field says `false`, some or all of the containing transactions might have already been executed.

```
function execute(uint256 proposalId) public payable {
     require(
         state(proposalId) == ProposalState.Queued,
         "Governor::execute: proposal can only be executed if it is queued"
     );
     Proposal storage proposal = proposals[proposalId];
     proposal.executed = true;
     for (uint256 i = 0; i < proposal.targets.length; i++) {
         timelock.executeTransaction.value(proposal.values[i])(
```

Figure 8.1: `Governor.sol#L173-L181`

The `Governor` execute function calls `Timelock.executeTransaction` for all the transactions within the `Proposal`.

```
function executeTransaction(
     address target,
     uint256 value,
     string memory signature,
     bytes memory data,
     uint256 eta
 ) public payable returns (bytes memory) {
     bytes32 txHash = keccak256(
         abi.encode(target, value, signature, data, eta)
     );
     require(
         queuedTransactions[txHash],
```

```
        "Timelock::executeTransaction: Transaction hasn't been queued."
    );
    require(
        getBlockTimestamp() >= eta,
        "Timelock::executeTransaction: Transaction hasn't surpassed time lock."
    );
    require(
        getBlockTimestamp() <= eta.add(GRACE_PERIOD),
        "Timelock::executeTransaction: Transaction is stale."
    );

    queuedTransactions[txHash] = false;

    bytes memory callData;

    if (bytes(signature).length == 0) {
        callData = data;
    } else {
        callData = abi.encodePacked(
            bytes4(keccak256(bytes(signature))),
            data
        );
    }

    // solium-disable-next-line security/no-call-value
    (bool success, bytes memory returnData) = target.call.value(value)(
        callData
    );
    require(
        success,
        "Timelock::executeTransaction: Transaction execution reverted."
    );
```

Figure 8.2: `Timelock.sol#L160-L203`

Anybody can call the `Timelock.executeTransaction` function to execute a specific transaction. If a transaction was already executed it will revert. If any of the transactions in a Proposal revert the entire Governor.execute call reverts.

If any of the transactions in a `Proposal` with multiple transactions have been executed separately, the `Governor.execute` function cannot be used to execute the remaining transactions, as the already executed one will revert. The only way to execute the remaining transactions is separately executing them through `Timelock.executeTransaction`. This also means that when one transaction has been separately executed, the `Proposal.executed` field will forever remain `false`.

A `Proposal` could contain multiple transactions that should be executed simultaneously to keep the contract functioning correctly. Executing the `Proposal` through `Governor.execute` would satisfy this requirement. However, only executing a specific transaction by directly executing it through `Timelock.executeTransaction` would break this requirement.

Exploit Scenario
A `Proposal` with three transactions that should be executed simultaneously has been queued and its `eta` has passed the delay time. Eve sees that the `Proposal` can be executed but notices that if only the second transaction is executed the contract will behave incorrectly and to her advantage. Eve calls `Timelock.executeTransaction` to execute the second transaction and uses the resulting state of the contract to her advantage.

Recommendation
Short term, only allow the `admin` to call `Timelock.executeTransaction`.

Long term, use property-based testing using Echidna to ensure the contract behaves as expected. Consider letting `Governor` inherit from `Timelock`. This would allow a lot of functions and code to be removed and significantly lower the complexity of these two contracts.

9. Proposals could allow Timelock.admin takeover

Severity: High
Type: Data Validation
Target: `Governor.sol, Timelock.sol`

Difficulty: High
Finding ID: TOB-OUSD-009

Description
The `Governor` contract contains special functions to let the `guardian` queue a transaction to change the `Timelock.admin`. However, a regular `Proposal` is also allowed to contain a transaction to change the `Timelock.admin`. This poses an unnecessary risk in that an attacker could create a `Proposal` to change the `Timelock.admin`.

```
function __queueSetTimelockPendingAdmin(
     address newPendingAdmin,
     uint256 eta
 ) public {
     require(
         msg.sender == guardian,
         "Governor::__queueSetTimelockPendingAdmin: sender must be gov guardian"
     );
     timelock.queueTransaction(
         address(timelock),
         0,
         "setPendingAdmin(address)",
```

Figure 9.1: `Governor.sol#L214-L225`

The `guardian` can queue a transaction to change the `pendingAdmin`.

```
function queueTransaction(
    address target,
    uint256 value,
    string memory signature,
    bytes memory data,
    uint256 eta
) public returns (bytes32) {
    require(
        msg.sender == admin,
        "Timelock::queueTransaction: Call must come from admin."
    );
```

Figure 9.2: `Timelock.sol#L115-L125`

If an attacker manages to become the `Timelock.admin` then the `Governor` could no longer call `Timelock.queueTransaction`. The only way out of this situation would be to redeploy the `Timelock` contract.

The `Governor` contract does not contain a function to update the `Timelock` contract. So also the `Governor` would need to be redeployed. This would also require all of the other contracts to update the governor address to the new `Timelock` contract.

Exploit Scenario

Eve creates a proposal with five transactions, one of which is a call to `setPendingAdmin` with an address controlled by Eve. The `guardian` doesn't notice this and queues the `Proposal`. Once the delay is passed Eve executes the `Proposal` and becomes the `pendingAdmin`. Eve calls `acceptAdmin` and is now the `admin` of the `Timelock`.

Recommendation

Short term, add a check that prevents `setPendingAdmin` to be included in a `Proposal`.

Long term, consider letting `Governor` inherit from `Timelock`. This would allow a lot of functions and code to be removed and significantly lower the complexity of these two contracts.

10. Reentrancy and untrusted contract call in `mintMultiple`

Severity: High
Difficulty: Low
Type: Data Validation
Finding ID: TOB-OUSD-010
Target: `VaultCore.sol`

Description

Missing checks and no reentrancy prevention allow untrusted contracts to be called from `mintMultiple`. This could be used by an attacker to drain the contracts.

```
function mintMultiple(
    address[] calldata _assets,
    uint256[] calldata _amounts
) external whenNotDepositPaused {
    require(_assets.length == _amounts.length, "Parameter length mismatch");

    uint256 priceAdjustedTotal = 0;
    uint256[] memory assetPrices = _getAssetPrices(false);
    for (uint256 i = 0; i < allAssets.length; i++) {
        for (uint256 j = 0; j < _assets.length; j++) {
            if (_assets[j] == allAssets[i]) {
                if (_amounts[j] > 0) {
                    uint256 assetDecimals = Helpers.getDecimals(
                        allAssets[i]
                    );
                    uint256 price = assetPrices[i];
                    if (price > 1e18) {
                        price = 1e18;
                    }
                    priceAdjustedTotal += _amounts[j].mulTruncateScale(
                        price,
                        10**assetDecimals
                    );
                }
            }
        }
    }
    // Rebase must happen before any transfers occur.
```

```
    if (priceAdjustedTotal > rebaseThreshold && !rebasePaused) {
        rebase(true);
    }

    for (uint256 i = 0; i < _assets.length; i++) {
        IERC20 asset = IERC20(_assets[i]);
        asset.safeTransferFrom(msg.sender, address(this), _amounts[i]);
    }

    oUSD.mint(msg.sender, priceAdjustedTotal);
    emit Mint(msg.sender, priceAdjustedTotal);

    if (priceAdjustedTotal >= autoAllocateThreshold) {
        allocate();
    }
}
```

Figure 10.1: `VaultCore.sol#L84-L127`

If an asset is not supported the first two loops will skip it. Likewise, if the amount is zero the first two loops will skip it. Compare this to the `mint` function which will revert if any of these two checks fail.

Unlike the first two loops, the third loop will not skip unsupported assets. This loop will call a function with the ERC20 `transferFrom` signature on each of the passed in asset addresses. An attacker could create a custom contract with such a function and it will be called. The attacker is free to do as he pleases within this function.

There are no reentrancy guards in the `VaultCore` contract and thus the above custom contract could call back into any of the `VaultCore` functions. Had there been reentrancy protection the attacker contract would not be able to call back into the `VaultCore` contract, severely limiting his abilities.

The third loop will transfer any assets into the `VaultCore` contract. However, only after the third loop is the corresponding OUSD minted. This creates a temporary imbalance between the assets transferred into `VaultCore`, and the minted OUSD. If an attacker contract is called inside this loop he could exploit this temporary imbalance.

Exploit Scenario

An attacker creates a custom contract containing a `transferFrom` function which calls `Vault.mint`. The attacker calls `mintMultiple` passing in USDT as the first asset and his custom contract as the second asset. The `mintMultiple` function will first transfer the USDT into the `VaultCore` contract, followed by calling the custom contract's `transferFrom` function. This function calls `Vault.mint` which triggers a rebase. Since the USDT already got transferred into the `VaultCore` contract, but no corresponding OUSD was minted, the imbalance will cause the rebase to function unexpectedly.

Recommendation
Short term, add checks that cause `mintMultiple` to revert if the amount is zero or the asset is not supported. Add a reentrancy guard to the `mint`, `mintMultiple`, `redeem`, and `redeemAll` functions.

Long term, make use of [Slither]() which will flag the reentrancy. Or even better, use [Crytic]() and incorporate static analysis checks into your CI/CD pipeline. Add reentrancy guards to all non-view functions callable by anyone. Make sure to always revert a transaction if an input is incorrect. Disallow calling untrusted contracts.

11. Off-by-one `minDrift/maxDrift` causes unexpected revert

Severity: Low
Type: Data Validation
Target: `MixOracle.sol`

Difficulty: High
Finding ID: TOB-OUSD-011

Description

The `MixOracle` contract contains a `minDrift` and `maxDrift` variable, indicating the min and max allowed drift of the price reported by the oracles. There is an off-by-one in the checks that make use of these variables. This will cause an error to be generated when the price is exactly the `minDrift` or `maxDrift`, even though the error indicates that the min/max value has been exceeded.

```
require(price < maxDrift, "Price exceeds max value.");
require(price > minDrift, "Price lower than min value.");
```

Figure 11.1: `MixOracle.sol#L128-L129`

```
require(price < maxDrift, "Price above max value.");
require(price > minDrift, "Price below min value.");
```

Figure 11.2: `MixOracle.sol#L179-L180`

Exploit Scenario

An oracle reports a price equal to the `minDrift`. The transaction reverts with a message that the price was below the `minDrift`.

Recommendation

Short term, update the checks to allow the `minDrift` and `maxDrift` values. Add unit tests to ensure the boundary values are allowed.

Long term, identify other places in the codebase that contain similar checks and ensure that boundary values are checked correctly. Make use of [Manticore] to symbolically verify the checks work as expected.

12. Unsafe last array element removal poses future risk

Severity: Undetermined
Type: Arithmetic
Target: `VaultAdmin.sol, MixOracle.sol`

Difficulty: High
Finding ID: TOB-OUSD-012

Description

Currently there are checks to prevent the removal of the last array element if there are no elements in the array. However, due to the contracts being upgradable, a future change might allow this to happen and cause specific functions to revert.

```
allStrategies[strategyIndex] = allStrategies[allStrategies.length -
                1];
allStrategies.length--;
```

Figure 12.1: `VaultAdmin.sol#L147-L149`

```
ethUsdOracles[i] = ethUsdOracles[ethUsdOracles.length - 1];
delete ethUsdOracles[ethUsdOracles.length - 1];
ethUsdOracles.length--;
```

Figure 12.2: `MixOracle.sol#L62-L64`

To remove the last array element the array length is decremented by one. If an array contains no elements and the length is decremented, an underflow will occur setting the array length to a very large number.

This would make the loops that iterate over these arrays revert either because of iterating too many times or the element not existing.

To prevent an underflow when removing the last array element Solidity added an array `pop()` method. This will remove the last item, decrease the length by one, and most importantly revert if the array is already of length zero.

Exploit Scenario

The contract is upgraded and a mistake has been made that allows to decrease the array length when there are no elements in the `ethUsdOracles` array. Due to this all functions that make use of `priceMin/priceMax` will revert as calling `tokEthPrice` on address zero causes a revert.

Recommendation

Short term, remove the last element of an array using `pop()`.

Long term, keep track and make use of new Solidity features that prevent common bugs.

13. Strategy targetWeight can be set for non-existent strategy

Severity: Low
Type: Data Validation
Target: `VaultAdmin.sol`

Difficulty: High
Finding ID: TOB-OUSD-013

Description

The `setStrategyWeights` function can be used to set the `targetWeight` of strategies that do not (yet) exist.

```
function setStrategyWeights(
    address[] calldata _strategyAddresses,
    uint256[] calldata _weights
) external onlyGovernor {
    require(
        _strategyAddresses.length == _weights.length,
        "Parameter length mismatch"
    );

    for (uint256 i = 0; i < _strategyAddresses.length; i++) {
        strategies[_strategyAddresses[i]].targetWeight = _weights[i];
    }

    emit StrategyWeightsUpdated(_strategyAddresses, _weights);
}
```

Figure 13.1: `VaultAdmin.sol#L173-L187`

There is no check to make sure the strategy exists. At the end an event is emitted indicating that for some (supposed) strategy contract the weight was set. This might confuse anybody monitoring the events emitted by this contract.

The `addStrategy` function will overwrite any existing strategy `targetWeight`. Setting the `targetWeight` for a non-existent strategy therefore has no effect, besides the incorrectly emitted event.

Exploit Scenario

Bob is monitoring the events emitted by the `VaultAdmin` contract. The `governor` calls the `setStrategyWeights` to set the weight for a non-existent strategy, causing the `StrategyWeightsUpdated` event to be emitted. Bob is confused as that strategy does not exist.

Recommendation
Short term, add a check that causes a revert if the strategy does not exist.

Long term, write a specification of each function and thoroughly test it with unit tests, fuzzing and symbolic execution.

14. Lack of minimum redeem value might lead to less return than expected

Severity: Medium
Type: Timing
Target: `VaultCore.sol`

Difficulty: High
Finding ID: TOB-OUSD-014

Description

The lack of a minimum redeem amount argument in the redeem functions could make a redeemer receive less assets than expected.

```
if (redeemFeeBps > 0) {
    uint256 redeemFee = _amount.mul(redeemFeeBps).div(10000);
    _amount = _amount.sub(redeemFee);
}
```

Figure 14.1: `VaultCore.sol#L579-L582`

The redeem fee is deducted from the redeem `_amount` inside the `_calculateRedeemOutputs` function.

An executable `Proposal` could exist to update the `redeemFeeBps` to a higher value. If somebody now calls `redeem`, while at the same time somebody executes the `Proposal`, the `redeem` call would return less funds than the caller expected.

To prevent such unexpected results a minimum expected return value could be added as an argument to the redeem functions. This would cause the `redeem` call to revert if the return value deviates too much from what the user expected. Preventing surprises for users calling `redeem` while the redeem fee is being changed.

Exploit Scenario

A `Proposal` to increase the `redeemFeeBps` exists and is executable. Bob calls `redeem` while at the same time Alice executes the `Proposal`. Bob receives less than what he expected due to the sudden increase of the redeem fee.

Recommendation

Short term, add a minimum expected redeem value argument to the redeem functions.

Long term, identify other functions that might be affected by a contract parameter change and add mitigations to protect users from such surprises.

15. withdraw allows redeemer to withdraw accidentally sent tokens

Severity: Low
Difficulty: Low
Type: Undefined Behavior
Finding ID: TOB-OUSD-015
Target: `AaveStrategy.sol`

Description

The `AaveStrategy.withdraw` function accidentally transfers the entire contract's token balance to the recipient, instead of the requested amount.

```
function withdraw(
    address _recipient,
    address _asset,
    uint256 _amount
) external onlyVault returns (uint256 amountWithdrawn) {
    require(_amount > 0, "Must withdraw something");
    require(_recipient != address(0), "Must specify recipient");

    IAaveAToken aToken = _getATokenFor(_asset);

    amountWithdrawn = _amount;
    uint256 balance = aToken.balanceOf(address(this));

    aToken.redeem(_amount);
    IERC20(_asset).safeTransfer(
        _recipient,
        IERC20(_asset).balanceOf(address(this))
    );

    emit Withdrawal(_asset, address(aToken), amountWithdrawn);
}
```

Figure 15.1: `AaveStrategy.sol#L45-L64`

The `AaveStrategy` contract is implemented to always pass-through all the ERC20 tokens to the recipient. However, someone might still accidentally transfer ERC20 tokens to this contract. In that case, the `withdraw` function will transfer them all to the `recipient`.

If somebody spots that there are accidentally sent tokens in the `AaveStrategy` contract they could withdraw them by redeeming (some of) their OUSD.

This function returns the `amountWithdrawn` variable, which might be incorrect. However, none of the calling functions use the returned value.

The `AaveStrategy` inherits from `InitializableAbstractStrategy`. This contract contains a function named transferToken which allows the Governor to extract accidentally sent tokens. This issue allows anybody to withdraw accidentally sent tokens instead of just the Governor.

Exploit Scenario
Eve spots somebody accidentally sending USDT tokens to the AaveStrategy contract. Eve calls `Vault.redeem`, which calls `AaveStrategy.withdraw` and transfers all of the USDT in the AaveStrategy contract to Bob.

Recommendation
Short term, transfer `amountWithdrawn` instead of the entire contract's ERC20 balance. Remove the return variable as the calling contract does not use it.

Long term, use Echidna to write properties that ensure ERC20 transfers are transferring the expected amount. Remove unused return variables in other functions throughout the codebase.

16. Variable shadowing from OUSD to ERC20

Severity: Low
Difficulty: Low
Type: Undefined Behavior
Finding ID: TOB-OUSD-016
Target: `OUSD.sol, @openzeppelin/contracts/token/ERC20/ERC20.sol`

Description

`OUSD` inherits from `ERC20`, but redefines the `_allowances` and `_totalSupply` state variables. As a result, access to these variables can lead to returning different values.

`OUSD` inherits from `InitializableToken`, which inherits from `ERC20`:

```
contract OUSD is Initializable, InitializableToken, Governable {
```

Figure 16.1: `OUSD.sol#L19`

```
contract InitializableToken is ERC20, InitializableERC20Detailed {
```

Figure 16.2: `InitializableToken.sol#L6`

Both `OUSD` and `ERC20` define `_allowances` and `_totalSupply`:

```
uint256 private _totalSupply;
[...]
mapping(address => mapping(address => uint256)) private _allowances;
```

Figure 16.3: `OUSD.sol#L31-L39`

```
mapping (address => mapping (address => uint256)) private _allowances;

uint256 private _totalSupply;
```

Figure 16.4: [`ERC20.sol#L34-L38`]

This shadowing leads the usage of these variables in `OUSD` and `ERC20` to refer to different variables, which can lead to unexpected behaviors.

We classified this issue as low severity, as currently all the functions in `ERC20` that rely on these variables are overridden in `OUSD`.

Exploit Scenario

The origin dollar team realizes that the `allowance(address, address)` function is already implemented in `ERC20`, and because it has the same code as in `OUSD`, the team decides to remove the `OUSD` version.

Recommendation

Short term, remove the shadowed variables (`_allowances` and `_totalSupply`) in `OUSD`.

Long term, use [Slither](https://github.com/crytic/slither) or subscribe to [Crytic.io](https://crytic.io/) to detect variables shadowing. Crytic catches the bug.

17. `VaultCore.rebase` functions have no return statements

Severity: Low
Difficulty: Low
Type: Undefined Behavior
Finding ID: TOB-OUSD-017
Target: `VaultCore.sol`

Description

`VaultCore.rebase()` and `VaultCore.rebase(bool)` return a `uint` but lack a return statement. As a result these functions will always return the default value, and are likely to cause issues for their callers.

Both `VaultCore.rebase()` and `VaultCore.rebase(bool)` are expected to return a `uint256`:

```
/**
 * @dev Calculate the total value of assets held by the Vault and all
 *         strategies and update the supply of oUSD
 */
function rebase() public whenNotRebasePaused returns (uint256) {
    rebase(true);
}

/**
 * @dev Calculate the total value of assets held by the Vault and all
 *         strategies and update the supply of oUSD
 */
function rebase(bool sync) internal whenNotRebasePaused returns (uint256) {
    if (oUSD.totalSupply() == 0) return 0;
    uint256 oldTotalSupply = oUSD.totalSupply();
    uint256 newTotalSupply = _totalValue();
    // Only rachet upwards
    if (newTotalSupply > oldTotalSupply) {
        oUSD.changeSupply(newTotalSupply);
        if (rebaseHooksAddr != address(0)) {
            IRebaseHooks(rebaseHooksAddr).postRebase(sync);
        }
    }
}
```

Figure 17.1: `VaultCore.sol#L292-L315`

`rebase()` does not have a return statement. `rebase(bool)` has one return statement in one branch (`return 0`), but lacks a return statement for the other paths. So both functions will always return zero.

As a result, a third-party code relying on the return value might not work as intended.

Exploit Scenario
Bob's smart contract uses `rebase()`. Bob assumes that the value returned is the amount of assets rebased. Its contract checks that the return value is always greater than zero. Since this function always returns 0, Bob's contract does not work.

Recommendation
Short term, add the missing return statement(s) or remove the return type in `VaultCore.rebase()` and `VaultCore.rebase(bool)`. Properly adjust the documentation as necessary.

Long term, use [Slither](#) or subscribe to [Crytic.io](#) to detect when functions are missing appropriate return statements. Crytic catches this bug type.

18. Multiple contracts are missing inheritances

Severity: Informational | Difficulty: Low
Type: Undefined Behavior | Finding ID: TOB-OUSD-018
Target: `OpenUniswapOracle.sol, IPriceOracle.sol, ChainlinkOracle.sol, RebaseHooks.sol, IRebaseHooks.sol`

Description
Multiple contracts are the implementation of their interfaces, but do not inherit from them. This behavior is error-prone and might lead the implementation to not follow the interface if the code is updated.

The contracts missing the inheritance are:

- `ChainlinkOracle` should inherit from `IPriceOracle`
- `OpenUniswapOracle` should inherit from `IPriceOracle`
- `RebaseHooks` should inherit from `IRebaseHooks`

Exploit Scenario
`IPriceOracle` is updated and one of its functions has a new signature. `ChainlinkOracle` is not updated. As a result, any call to the updated function using `ChainlinkOracle` will fail.

Recommendation
Short term, ensure `ChainlinkOracle and OpenUniswapOracle` inherits from `IPriceOracle`, and that `RebaseHooks` inherits from `IRebaseHooks`.

Long term, subscribe to crytic.io. Crytic catches the bug.

19. Lack of return value checks can lead to unexpected results

Severity: High
Difficulty: Low
Type: Undefined Behavior
Finding ID: TOB-OUSD-019
Target: Several contracts

Description
Several function calls do not check the return value. Without a return value check, the code is error-prone, which may lead to unexpected results.

The functions missing the return value check include:

- `CompoundStrategy.liquidate() (strategies/CompoundStrategy.sol#73-87)` ignores return value by `cToken.redeem(cToken.balanceOf(address(this))) (strategies/CompoundStrategy.sol#78)`
- `InitializableAbstractStrategy.transferToken(address,uint256) (utils/InitializableAbstractStrategy.sol#190-195)` ignores return value by `IERC20(_asset).transfer(governor(),_amount) (utils/InitializableAbstractStrategy.sol#194)`
- `VaultAdmin.transferToken(address,uint256) (vault/VaultAdmin.sol#235-240)` ignores return value by `IERC20(_asset).transfer(governor(),_amount) (vault/VaultAdmin.sol#239)`
- `VaultAdmin._harvest(address) (vault/VaultAdmin.sol#266-298)` ignores return value by `IUniswapV2Router(uniswapAddr).swapExactTokensForTokens(rewardTokenAmount,uint256(0),path,address(this),now.add(1800)) (vault/VaultAdmin.sol#288-294)`
- `Governor._queueOrRevert(address,uint256,string,bytes,uint256) (governance/Governor.sol#157-171)` ignores return value by `timelock.queueTransaction(target,value,signature,data,eta) (governance/Governor.sol#170)`
- `Governor.execute(uint256) (governance/Governor.sol#173-190)` ignores return value by `timelock.executeTransaction.value(proposal.values[i])(proposal.targets[i],proposal.values[i],proposal.signatures[i],proposal.calldatas[i],proposal.eta) (governance/Governor.sol#181-187)`
- `Governor.__queueSetTimelockPendingAdmin(address,uint256) (governance/Governor.sol#214-229)` ignores return value by `timelock.queueTransaction(address(timelock),0,setPendingAdmin(address),abi.encode(newPendingAdmin),eta) (governance/Governor.sol#222-228)`
- `Governor.__executeSetTimelockPendingAdmin(address,uint256) (governance/Governor.sol#231-246)` ignores return value by

`timelock.executeTransaction(address(timelock),0,setPendingAdmin(address),abi.encode(newPendingAdmin),eta) (governance/Governor.sol#239-245)`

- `VaultCore._redeem(uint256) (vault/VaultCore.sol#140-182)` ignores return value by `strategy.withdraw(msg.sender,allAssets[i],outputs[i]) (vault/VaultCore.sol#163)`
- `VaultCore._allocate() (vault/VaultCore.sol#207-290)` ignores return value by `strategy.deposit(address(asset),allocateAmount) (vault/VaultCore.sol#261)`
- `VaultCore.rebase(bool) (vault/VaultCore.sol#304-315)` ignores return value by `oUSD.changeSupply(newTotalSupply) (vault/VaultCore.sol#310)`

Exploit Scenario

The `VaultCore._redeem` function calls `CompoundStrategy.withdraw`. For some reason there are no tokens to redeem and zero is returned. Inside `VaultCore._redeem` the return value is not checked and the code will continue to burn the OUSD of the user.

Recommendation

Short term, check the return value of all calls mentioned above.

Long term, subscribe to crytic.io to catch missing return checks. Crytic identifies this bug type automatically.

20. External calls in loop can lead to denial of service

Severity: High
Type: Denial of Service
Target: Several Contracts

Difficulty: Medium
Finding ID: TOB-OUSD-020

Description
Several function calls are made in unbounded loops. This pattern is error-prone as it can trap the contracts due to the gas limitations or failed transactions.

For example, `AaveStrategy` has several loops that iterate over the `assetsMapped` items, including `safeApproveAllTokens`:

```
function safeApproveAllTokens() external onlyGovernor {
    uint256 assetCount = assetsMapped.length;
    address lendingPoolVault = _getLendingPoolCore();
    // approve the pool to spend the bAsset
    for (uint256 i = 0; i < assetCount; i++) {
        address asset = assetsMapped[i];
        // Safe approval
        IERC20(asset).safeApprove(lendingPoolVault, 0);
        IERC20(asset).safeApprove(lendingPoolVault, uint256(-1));
    }
}
```

Figure 20.1: `AaveStrategy.sol#L114-L124`

`assetsMapped` is an unbounded array that can only grow. `safeApproveAllTokens` can be trapped if:

- A call to an asset fails (for example, the asset is paused)
- Items in `assetsMapped` increases the gas cost beyond a certain limit

Similar patterns exist in:

- CompoundStrategy.liquidate()
- CompoundStrategy.safeApproveAllTokens()
- MixOracle.priceMin(string)
- MixOracle.priceMax(string)
- RebaseHooks.postRebase(bool)
- AaveStrategy.liquidate()
- Governor.execute(uint256)

Exploit Scenario
Over time, the governor adds dozens of assets in `assetsMapped`. As a result `safeApproveAllTokens` is no longer callable.

Recommendation
Short term, review all the loops mentioned above and either:

- allow iteration over part of the loop, or
- remove elements.

Long term, subscribe to crytic.io to review external calls in loops. Crytic catches bugs of this type.

21. No events for critical operations

Severity: Informational
Type: Auditing and Logging
Target: Several contracts

Difficulty: Low
Finding ID: TOB-OUSD-021

Description
Several critical operations do not trigger events. As a result, it will be difficult to review the correct behavior of the contracts once deployed.

Critical operations that would benefit from triggering events include:

- `MixOracle.sol#L35-L41`
  - Lack of events when setting `(min|max)Drift`
- `VaultAdmin.sol#L33-L35`
  - Lack of events when setting price provider
- `VaultAdmin.sol#L41-L43`
  - Lack of events when setting redeem fee bps
- `VaultAdmin.sol#L50-L52`
  - Lack of events when setting `vaultBuffer`
- `VaultAdmin.sol#L59-L64`
  - Lack of events when setting auto allocate threshold
- `VaultAdmin.sol#L71-L73`
  - Lack of events when setting rebase threshold
- `VaultAdmin.sol#L80-L82`
  - Lack of events when setting rebase hooks contract address
- `VaultAdmin.sol#L89-L91`
  - Lack of events when setting uniswap contract address
- `VaultAdmin.sol#L196-L214`
  - Lack of events when (un)pausing `rebase`
- `ChainlinkOracle.sol#L34-L44`
  - Lack of events when registering a feed
- `MixOracle.sol#L47-L69`
  - Lack of events when (de)registering eth-usd oracles
- `MixOracle.sol#L76-L84`
  - Lack of events when setting token-eth oracles
- `OpenUniswapOracle.sol#L41-L46`
  - Lack of events when registering eth price oracles
- `OpenUniswapOracle.sol#L48-L72`
  - Lack of events when registering a token pair

Users and blockchain monitoring systems will not be able to easily detect suspicious behaviors without events.

Exploit Scenario
Eve compromises the `VaultAdmin` contract and sets `redeemFeeBps` to a higher value. Bob does not notice the compromise and suddenly has to pay a higher redeem fee.

Recommendation
Short term, add events for all critical operations. Events help to monitor the contracts and detect suspicious behavior.

Long term, consider using a blockchain monitoring system to track any suspicious behavior in the contracts. The system relies on the correct behavior of several contracts. A monitoring system which tracks critical events would allow quick detection of any compromised system components.

22. OUSD allows users to transfer more tokens than expected

Severity: High
Difficulty: High
Type: Data Validation
Finding ID: TOB-OUSD-022
Target: `OUSD.sol`

Description

Under certain circumstances, the OUSD contract allows users to transfer more tokens than the ones they have in their balance.

A user or external contract trying to transfer one token more than its balance will expect that transfer to revert or the transfer to return false. However, after executing the following sequence of transactions, user1 will be allowed to transfer one token more than its current balance.

```
initialize("", "", vault)
mint(user1, 333333333333333333333333333333333333333333333333333333)
mint(user2, 333333333333333333333333333333333333333333333333333333)
mint(user3, 333333333333333333333333333333333333333333333333333333)
changeSupply(15)
mint(user1, 16)
```

Figure 22.1: `Sequence of transactions to break a system invariant`

This issue seems to be caused by a rounding issue when the `creditsDeducted` is calculated and subtracted:

```
function _executeTransfer(
    address _from,
    address _to,
    uint256 _value
) internal {
    bool isNonRebasingTo = _isNonRebasingAccount(_to);
    bool isNonRebasingFrom = _isNonRebasingAccount(_from);


    // Credits deducted and credited might be different due to the
    // differing creditsPerToken used by each account
    uint256 creditsCredited = _value.mulTruncate(_creditsPerToken(_to));
    uint256 creditsDeducted = _value.mulTruncate(_creditsPerToken(_from));
```

```
    _creditBalances[_from] = _creditBalances[_from].sub(
        creditsDeducted,
        "Transfer amount exceeds balance"
    );
    _creditBalances[_to] = _creditBalances[_to].add(creditsCredited);
    ...
```

Figure 22.2: `OUSD.sol#L154-L171`

Exploit Scenario
Eve interacts with the OUSD token, trying to transfer more tokens that she has. Instead of failing, the transaction succeeds, allowing it to credit more tokens than expected to another account.

Recommendation
Short term, make sure the balance is correctly checked before performing all the arithmetic operations. This will make sure it does not allow to transfer more than expected.

Long term, use [Echidna](https://github.com/crytic/echidna) to write properties that ensure ERC20 transfers are transferring the expected amount.

23. OUSD total supply can be arbitrary, even smaller than user balances

Severity: High
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-OUSD-023
Target: `OUSD.sol`

Description

The `OUSD` token contract allows users to opt out of rebasing effects. At that point, their exchange rate is "fixed", and further rebases will not have an impact on token balances (until the user opts in).

The `rebaseOptOut` is a public function that any account can call to be removed from the non-rebasing exception list.

```
function rebaseOptOut() public {
    require(!_isNonRebasingAccount(msg.sender), "Account has not opted in");


    // Increase non rebasing supply
    nonRebasingSupply = nonRebasingSupply.add(balanceOf(msg.sender));
    // Increase non rebasing credits
    nonRebasingCredits = nonRebasingCredits.add(
        _creditBalances[msg.sender]
    );


    // Set fixed credits per token
    nonRebasingCreditsPerToken[msg.sender] = rebasingCreditsPerToken;


    // Decrease rebasing credits, total supply remains unchanged so no
    // adjustment necessary
    rebasingCredits = rebasingCredits.sub(_creditBalances[msg.sender]);


    // Mark explicitly opted out of rebasing
    rebaseState[msg.sender] = RebaseOptions.OptOut;
}
```

Figure 23.1: `OUSD.sol#450-469`

However, calling `changeSupply` changes the `_totalSupply`, but not balances of user's that have opted out using the `rebaseOptOut`. As a result, it can happen that `_totalSupply` and balance of a user differ by an arbitrary amount. Put another way, the contract does not satisfy the common EIP-20 invariant that for all accounts, `balanceOf(x) <= totalSupply()`.

Since providing the option to opt out is part of the design of the contract, this issue is difficult to remedy.

Exploit Scenario
A third party contract assumes that `totalSupply` is greater than a user's balance. It is not and that can lead to unforeseen consequences.

Recommendation
Short term, we would advise making clear all common invariant violations for users and other stakeholders.

Long term, we would recommend designing the system in such a way to preserve as many commonplace invariants as possible.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program
External Interaction	Related to interactions with external programs
Standards	Related to complying with industry standards and best practices

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important

Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components.
Arithmetic	Related to the proper use of mathematical operations and semantics.
Assembly Use	Related to the use of inline assembly.
Centralization	Related to the existence of a single point of failure.
Upgradeability	Related to contract upgradeability.
Function Composition	Related to separation of the logic into functions with clear purpose.
Front-Running	Related to resilience against front-running.
Key Management	Related to the existence of proper procedures for key generation, distribution, and access.
Monitoring	Related to use of events and monitoring procedures.
Specification	Related to the expected codebase documentation.
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.).

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- `AaveStrategy.sol#L56`
  - `balance` is an unused local variable
- `OpenUniswapOracle.sol#L219-L263`
  - Remove debug functions in a production contract

D. Token Integration Checklist

The following checklist provides recommendations when interacting with arbitrary tokens. Every unchecked item should be justified and its associated risks understood.

This checklist is maintained on the internet at crytic/building-secure-contracts. Please see the version on Github for the most up-to-date recommendations.

For convenience, all Slither utilities can be run directly on a token address, such as:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken
```

To follow this checklist, you will want to have this output from Slither for the token:

```
- slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
- slither [target] --print human-summary
- slither [target] --print contract-summary
- slither-prop . --contract ContractName # requires configuration, and use of
Echidna and Manticore
```

General Security Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (aka "level of effort"), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on blockchain-security-contacts.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

ERC Conformity

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use slither-check-erc to review that:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and might not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. If this is the case, ensure the value returned is below 255.

- ❑ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.
- ❑ **The token is not an ERC777 token and has no external function call in `transfer` and `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use slither-prop to review that:

- ❑ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests, then check the properties with Echidna and Manticore.

Finally, there are certain characteristics that are difficult to identify automatically. Review for these conditions by hand:

- ❑ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❑ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest might be trapped in the contract if not taken into account.

Contract Composition

- ❑ **The contract avoids unneeded complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❑ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❑ **The contract has only a few non–token-related functions.** Non–token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.
- ❑ **The token only has one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g. `balances[token_address][msg.sender]` might not reflect the actual balance).

Owner privileges

- ❑ **The token is not upgradeable.** Upgradeable contracts might change their rules over time. Use Slither's `human-summary` printer to determine if the contract is upgradeable.
- ❑ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.

- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pauseable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams, or that reside in legal shelters should require a higher standard of review.

Token Scarcity

Reviews for issues of token scarcity requires manual review. Check for these conditions:

- ❏ **No user owns most of the supply.** If a few users own most of the tokens, they can influence operations based on the token's repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange can compromise the contract relying on the token.
- ❏ **Users understand the associated risks of large funds or flash loans.** Contracts relying on the token balance must carefully take in consideration attackers with large funds or attacks through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

E. Fix Log

Origin protocol addressed most issues in their codebase as a result of our assessment. Each of the fixes provided was checked by Trail of Bits on the week of December 14th.

#	Title	Type	Severity	Status
1	Invalid vaultBuffer could revert allocate	Data Validation	Low	Fixed (f741c68)
2	OUSD.changeSupply should require rebasingCreditsPerToken > 0	Data Validation	High	Fixed (#376)
3	SafeMath is recommended in OUSD._executeTransfer	Data Validation	Informational	Fixed (#375)
4	Transfers could silently fail without safeTransfer	Undefined Behavior	Informational	Fixed (#378)
5	Proxies are only partially EIP-1967-compliant	Standards	Informational	Not fixed
6	Queued transactions cannot be cancelled	Access Controls	High	Fixed (#372)
7	Unused code could cause problems in future	Undefined Behavior	Undetermined	Fixed (#383, #384)
8	Proposal transactions can be executed separately and block Proposal.execute call	Undefined Behavior	High	Fixed (#372, #432)
9	Proposals could allow Timelock admin takeover	Data Validation	High	Not fixed (#385, #432)
10	Reentrancy and untrusted contract call in mintMultiple	Data Validation	High	Fixed (#380)
11	Off-by-one minDrift/maxDrift causes unexpected revert	Data Validation	Low	Fixed (#373)
12	Unsafe last array element removal poses future risk	Arithmetic	Undetermined	Fixed (#374)
13	Strategy targetWeight can be set for non-existent strategy	Data Validation	Low	Fixed (#368)

14	Lack of minimal redeem value might lead to less return than expected	Timing	Medium	Fixed (#390)
15	withdraw allows redeemer to withdraw accidentally sent tokens	Undefined Behavior	Low	Fixed (#377)
16	Variable shadowing from OUSD to ERC20	Undefined Behavior	Low	Fixed (#392)
17	VaultCore.rebase functions have no return statements	Undefined Behavior	Low	Fixed (90c945d)
18	Multiple contracts are missing inheritances	Undefined Behavior	Informational	Fixed (#381, #383, #384, #449)
19	Lack of return value checks can lead to unexpected results	Undefined Behavior	Undetermined	Fixed (#387, 9d3b08f)
20	External calls in loop can lead to denial of service	Denial of Service	High	Fixed (#388)
21	No events for critical operations	Auditing and Logging	Informational	Fixed (#382, #384, #450)
22	OUSD allows users to transfer more tokens than expected	Data Validation	High	Partially fixed (#412)
23	OUSD._totalSupply can be arbitrary, even smaller than user balances	Data Validation	High	Partially fixed (153bd8a, a0d61d3)

Detailed fix log

TOB-OUSD-001: Invalid vaultBuffer could revert allocate
Fixed. A check that `_vaultBuffer >= 0` is redundant, but harmless.

TOB-OUSD-003: SafeMath is recommended in OUSD._executeTransfer
Fixed. This issue has been solved by removing the `nonRebasingCredits` storage variable completely. While this is a superset of the changes we proposed, we don't think it introduces any new issues.

TOB-OUSD-009: Proposals could allow Timelock admin takeover
Not fixed. It is still possible to call the `setPendingAdmin` function from inside a proposal, e.g. by leaving the `signature` empty and pasting the method id into the `calldata`. In that case, the function `Timelock.executeTransaction` ignores (the hash of) the `signature`, and only the `calldata`, containing the method id, will be forwarded. Note that there are other methods, e.g. by finding a partial hash collision of the first byte, and placing the last three bytes of the method id into `calldata`.

Also note that the `__queueSetTimelockPendingAdmin` function has been removed from the `Governor` contract, so if this issue is fixed, it will not be possible to call `setPendingAdmin` and hence change the admin.

Both issues can be solved by changing the authorization of `setPendingAdmin` from `address(this)` to `onlyAdmin`.

TOB-OUSD-010: Reentrancy and untrusted contract call in mintMultiple
Fixed. This issue has been fixed by checking that all assets are supported and adding a reentrancy guard. However, there are multiple other changes in this PR that are beyond the scope of this review.

TOB-OUSD-014: Lack of minimal redeem value might lead to less return than expected
Fixed. A minimum redeem amount has been implemented. We recommend documenting that the function expects a value scaled to 18 decimal places, even though coins like USDC only have 6.

TOB-OUSD-022: OUSD allows users to transfer more tokens than expected
Partially fixed. While PR [#412] adds the checks to `transfer` and `transferFrom`, we think more tokens than owned can still be burned which is a DoS vector through totalSupply underflow. The supply cap is ineffective as any initial mint amount (123456, 333333, etc) still triggers the rounding error.

TOB-OUSD-023: OUSD._totalSupply can be arbitrary, even smaller than user balances
Partially fixed. We don't think the changes fully address the issue, but the `OUSD` contract now has a warning that it doesn't satisfy common EIP-20 invariants due to its design, hence we believe Origin Protocol accepts the risk.