Security assessment of PR #1239

Rappie, March 8th, 2023

Introduction
This document describes the security assessment of Pull Request #1239 *"Cleanup OUSD rebasing/nonrebasing accounting changes"* in the `OriginProtocol/origin-dollar` Github repository.

The assessment is performed by Rappie (twitter: @rappenstein2), between February and March of 2023.

Scope

The plan is to launch a fuzzing campaign that tests all functions in the `OUSD` contract that underwent code changes between the PR and `main` branches. Our goal is to uncover any potential bugs that may have arisen from the updated code.

Additionally, we intend to study the Spherax USDs hack and replicate it in the OUSD code by making minor modifications. This exercise will support us in assessing the security and resilience of the current code structure.

Lastly, we will conduct a quick manual review of the code to identify any potential problems that may not be detected by the fuzzing process alone.

Fuzzing results

The following functions have been tested:
- `rebaseOptIn`
- `rebaseOptOut`
- `_ensureMigrationToNonRebasing` (indirectly)

Results
Test	Result
Account balance should remain the same after opting in	Failed
Account balance should not increase when opting in	Passed
Account balance should remain the same after opting out	Passed
Account balance should remain the same when a smart contract auto converts	Passed

| Test `revert` behavior of opting in | Failed |
| Test `revert` behavior of opting out | Passed |

Analysis
Let's examine the failed tests more closely.

Account balance should remain the same after opting in
The reasons for the difference in balance before and after opting in are well-known and deemed acceptable. They do not cause any known problems in practice.

Some observations:
1. `rebaseOptIn` is known to have rounding errors
2. These errors always round down. Rounding up never occurs
3. The maximum amount of rounding is `1e18-1`, which is an acceptable amount

Test revert behavior of opting in
The reasons for the reverts in `rebaseOptIn` are well-known and deemed acceptable. They do not cause any known problems in practice.

The following scenarios are known to cause reverts:

- An overflow can happen with large amounts of credits and a very large `_rebasingCreditsPerToken` value. This scenario is currently impossible in practice because the implementation of `rebase` only supports decreasing `_rebasingCreditsPerToken`.

- An underflow can happen because of a known bug in the `burn` function. This bug can cause the total supply to decrease while the balance stays the same. This in turn can trigger a revert in `rebaseOptIn` while subtracting the balance from the total non-rebasing supply. The differences caused by `burn` are too small to be any cause for concern in practice.

Further investigation
It was noticed that many rounding errors are happening because of fuzzing the `changeSupply` function without bounds. To fix this issue, a helper function was created that adjusts the total supply by a (small) percentage. This modification made most of the rounding problems go away, and we didn't find any new previously hidden issues.

We also tested the code in the `main` branch with the same tests used for the PR. No noticeable differences between the new code in the PR were found.

Conclusion
No previously unknown issues have been discovered.

Spherax USDs Hack Analysis

To examine the hack, we begin by analyzing the code. The contract has been deployed on Arbitrum at address `0x97A7E6Cf949114Fe4711018485D757b9c4962307`. Since the code isn't validated on etherscan, we'll need to decompile it.

After extensive analysis, we've determined the primary reasons for the hack. In essence, the complex balance system combined with the migration logic being disorganized can be attributed to being the cause of the hack.

Let's look at this in more detail and attempt to replicate the same issues by incorporating comparable logic into the OUSD code.

USDs contract
Please be aware that due to the decompilation of the code, some function and variable names may differ from the original.

Rebase state is not updated during migrations
The `_rebaseState[account]` mapping is utilized to keep track of the contract's state as "not set", "opted in", or "opted out". This mapping is updated when the user chooses to opt in or out but not during migrations.

Because of this, in certain areas during the migration, a different check is performed:
```solidity
if (_nonRebasingCreditsPerToken[account] == 0) {
```

Migration logic code spread across multiple locations
The same migration logic code is used in at least two functions, namely:
- `rebaseOptOut`
- `isNonRebasing`

This increases the possibility of unintentionally generating minor differences between the two locations, especially if you're not familiar with the code.

State changes to account before retrieving balance
Take a look at the following code:
```solidity
if (_nonRebasingCreditsPerToken[account] == 0) {
        _nonRebasingCreditsPerToken[account] = 1;

        if (_balanceOf[account] > 0) {
```

```
                balance = getCredits(account);
                _nonRebasingSupply += balance;
                _balanceOf[account] = balance;
        }
}
```

The order of things here causes `getCredits` to return a very wrong result. Calling `getCredits(account)` should be performed **before** `_nonRebasingCreditsPerToken[account]` is updated.

Credit balance used for two separate accounting systems
The credit balance is used differently depending on whether or not the account is rebasing. This works as follows:
- If the account is rebasing, the credit balance is multiplied by the "credits per token" multiplier.
- If the account is non-rebasing, the credit balance is used as-is.

This approach forces the inclusion of extra logic when retrieving an account's balance. Additionally, during migration, it is necessary to update the balance directly instead of just the multiplier.

Having these separate accounting systems sharing the same state variable appears to be the primary issue that led to the chain of events resulting in the hack. It created technical debt in the system, making it susceptible to small human errors with significant consequences.

OUSD contract
Now that we've identified the primary problems in the USDs hack, let's explore how we can make minor modifications to the OUSD code to replicate the bug.

Rebase state is not updated during migrations
No adjustments to the code are currently required. More on this later.

Migration logic spread across multiple locations
To simulate this, we can create a copy of the `_ensureMigrationToNonRebasing` function and substitute the call from `_isNonRebasingAccount` with a call to this new function. We will name this function `_ensureMigrationToNonRebasingWithBug`.

```solidity
function _isNonRebasingAccount(address _account) public returns (bool) {
        ...
        if (isContract && rebaseState[_account] == RebaseOptions.NotSet) {
                _ensureMigrationToNonRebasingWithBug(_account);
```

```
        }
        ...
}

function _ensureMigrationToNonRebasingWithBug(address _account) internal {
        ...
```

State changes to account before retrieving balance
This can be achieved by simply reordering the lines in `_ensureMigrationToNonRebasingWithBug`.

```solidity
function _ensureMigrationToNonRebasingWithBug(address _account) internal {
        ...

        // Deliberately change account state before retrieving balance
        // related information.
        nonRebasingCreditsPerToken[_account] = _rebasingCreditsPerToken;

        // No changes here
        uint256 oldCredits = _creditBalances[_account];

        ...
}
```

Credit balance used for two separate accounting systems
Unlike USDs, OUSD does not employ two distinct accounting systems. However, we can replicate this behavior by intentionally introducing a bug that alters (increases) the account balance based on its state.

```solidity
function _ensureMigrationToNonRebasingWithBug(address _account) internal {
        ...

        // Simulate USDs' system where credits need to be converted and up-
        // dated during the migration.
        if (nonRebasingCreditsPerToken[_account] != 0) {
                // Increase the credits in some arbitrary way
                uint256 newCredits = oldCredits * _rebasingCreditsPerToken;
                _creditBalances[_account] = newCredits;
```

```
        }

        ...
}
```

Result
The final code looks like this:
```solidity
function _isNonRebasingAccount(address _account) public returns (bool) {
        // bool isContract = Address.isContract(_account);
        bool isContract = accountIsContract[_account]; // make it fuzzable
        if (isContract && rebaseState[_account] == RebaseOptions.NotSet) {
                _ensureMigrationToNonRebasingWithBug(_account);
        }
        return nonRebasingCreditsPerToken[_account] > 0;
}

function _ensureMigrationToNonRebasingWithBug(address _account) internal {
        // No changes here
        if (nonRebasingCreditsPerToken[_account] != 0) {
                return;
        }
        if (_creditBalances[_account] == 0) {
                nonRebasingCreditsPerToken[_account] = 1e27;
                return;
        }

        // Deliberately change account state before retrieving balance
        // related information.
        nonRebasingCreditsPerToken[_account] = _rebasingCreditsPerToken;

        // No changes here
        uint256 oldCredits = _creditBalances[_account];

        // Simulate USDs' system where credits need to be converted and up-
        // dated during the migration.
        if (nonRebasingCreditsPerToken[_account] != 0) {
                // Increase the credits in some arbitrary way
                uint256 newCredits = oldCredits * _rebasingCreditsPerToken;
                _creditBalances[_account] = newCredits;
        }
```

```
        // No changes here
        nonRebasingSupply = nonRebasingSupply.add(balanceOf(_account));
        _rebasingCredits = _rebasingCredits.sub(oldCredits);
}
```

Fuzzing the OUSD contract

EOA vs. Contract
Next, we will attempt to trigger the bug by fuzzing with Echidna. Before we can do that, we need to make one additional modification to the code.

As part of the hacking process, the hacker "flipped" an account's state from an EOA to a Contract. The USDs hacker accomplished this using a Gnosis Safe. We will simulate this process by introducing a new (fuzzable) state variable called `accountIsContract`. We can now replace the code in `_isNonRebasingAccount` as follows:

```solidity
// bool isContract = Address.isContract(_account);
bool isContract = accountIsContract[_account]; // make it fuzzable
if (isContract && rebaseState[_account] == RebaseOptions.NotSet) {
  ...
```

Helper functions
We will define the following helper functions to be used by Echidna:
```solidity
function makeAccountEOA(address account) public {
        accountIsContract[account] = false;
}

function makeAccountContract(address account) public {
        accountIsContract[account] = true;
}
```

Invariant
We will use an invariant to check whether the total balance has increased. Without being able to mint more tokens, this should normally be impossible to achieve.

```solidity
```

```
constructor() public {
        initialTotalBalance = getTotalBalance();
}

function invariantTotalBalance() public {
        assert(initialTotalBalance >= getTotalBalance());
}
```

Result
As you can see below, Echidna finds the bug within seconds. It seems to favor using `rebaseOptIn()` to trigger the migration. However, with some guidance, it can find the exact same steps used in the USDs hack by using `transfer` instead of `rebaseOptIn`.

Output, with some minor modifications to clean up the result:
```
invariantTotalBalance(): failed!💥 Call sequence
        makeAccountContract()
        rebaseOptIn()
        invariantTotalBalance()
```

Finally, we tried running the original fuzzing campaign and added just the part that makes it possible to flip an account between being an EOA and a Contract. No new issues were discovered.

Conclusion
By fuzzing the code with the EOA vs. Contract "flipping" technique, we can conclude with a high degree of certainty that the code in the PR does not contain any bugs comparable to the USDs hack.

Numerous illogical steps were necessary to alter the code and make it vulnerable to the bug.

The present OUSD codebase is set up in such a manner that the probability of a human error resulting in a bug similar to the one that affected USDs is minimal.

Manual code review

The manual review did not yield any important results besides a few minor recommendations.

Typing errors

A couple of typing error corrections have been suggested as comments in the pull request on Github.

Example:
```solidity
// Mark explicitly opted out of rebasing
rebaseState[msg.sender] = RebaseOptions.OptIn;
```

This should be: "Mark explicitly **opted in to** rebasing"

Updating rebase state during migration
Consider updating the `_rebaseState` variable while performing the migration in `_ensureMigrationToNonRebasing` to make sure it reflects the correct state as soon as it is known.

This suggestion has been discussed with the OUSD team, and it may not be preferred as one could argue that the `_rebaseState` variable should represent user intent.

Recommendations:
- Update the rebasing state as soon as it is known.
- If you prefer to use the variable to represent intent, consider renaming it to something more appropriate to make clear that it cannot (always) be relied upon to determine the account's state.

Opting in/out should emit Events
Consider having the following actions emit events:
- Opting in
- Opting out

Automatically migrating a Contract account to non-rebasing might also be a good candidate for adding an event.

References & Acknowledgements
Origin Dollar Pull Request *"Cleanup OUSD rebasing/nonrebasing accounting changes"*
https://github.com/OriginProtocol/origin-dollar/pull/1239

Spherax USDs: A recently compromised fork of OUSD
-
https://github.com/SunWeb3Sec/DeFiHackLabs#20230203---spherax-usds---balance-recalculation-bug
- https://medium.com/sperax/usds-feb-3-exploit-report-from-engineering-team-9f0fd3cef00c
- https://twitter.com/danielvf/status/1621965412832350208

-
https://phalcon.blocksec.com/tx/arbitrum/0xfaf84cabc3e1b0cf1ff1738dace1b2810f42d98baeea17b146ae032f0bdf82d5

Echidna - Smart Contract Fuzzer
https://github.com/crytic/echidna

Dedaub Smart Contract Bytecode Decompiler
https://library.dedaub.com/decompile

ChatGPT - Utilized as a resource to assist in writing this report
https://chat.openai.com/chat

Many thanks to DanielVF for the support and feedback

Disclaimer

The information, advice, or services provided by me (Rappie) were based on my best knowledge and abilities at the time of delivery. However, I cannot guarantee the accuracy, completeness, or usefulness of the information provided, nor can I be held liable for any damages or losses resulting from the use or reliance on such information or services. Any actions taken based on the information or advice provided are done so at the sole discretion and risk of the individual or organization involved. This disclaimer serves to clarify that while I strive to provide helpful and accurate information or services, there are no legal guarantees or warranties provided.