Angle Protocol - Invitational Findings & Analysis Report

2023-08-09

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Angle Protocol smart contract system written in Solidity. The audit took place between June 28—July 7 2023.

Following the C4 audit, 3 wardens (Lambda, auditor0517, and Jeiwan) reviewed the mitigations for all identified issues; the mitigation review report is appended below the audit report.

Wardens

In Code4rena's Invitational audits, the competition is limited to a small group of wardens; for this audit, 5 wardens contributed reports to the Angle Protocol:

1. auditor0517
2. Jeiwan
3. Lambda
4. __141345__
5. Udsen

This audit was judged by hansfriese.

Final report assembled by PaperParachute and liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 10 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 7 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 4 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 2 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Angle Protocol repository, and is composed of 24 smart contracts written in the Solidity programming language and includes 2276 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges

- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (3)

[H-01] Possible reentrancy during redemption/swap

Submitted by auditor0517, also found by Lambda

Redeemers might charge more collaterals during redemption/swap by the reentrancy attack.

Proof of Concept

Redeemers can redeem the agToken for collaterals in `Redeemer` contract and `_redeem()` burns the agToken and transfers the collaterals.

```
        function _redeem(
            uint256 amount,
            address to,
            uint256 deadline,
            uint256[] memory minAmountOuts,
            address[] memory forfeitTokens
        ) internal returns (address[] memory tokens, uint256[] memor
            TransmuterStorage storage ts = s.transmuterStorage();
            if (ts.isRedemptionLive == 0) revert Paused();
            if (block.timestamp > deadline) revert TooLate();
            uint256[] memory subCollateralsTracker;
            (tokens, amounts, subCollateralsTracker) = _quoteRedempt
            // Updating the normalizer enables to simultaneously anc
            // of stablecoins issued from each collateral without ha
            _updateNormalizer(amount, false);

            IAgToken(ts.agToken).burnSelf(amount, msg.sender); //@au

            address[] memory collateralListMem = ts.collateralList;
            uint256 indexCollateral;
```

```
        for (uint256 i; i < amounts.length; ++i) {
            if (amounts[i] < minAmountOuts[i]) revert TooSmallAm
            // If a token is in the `forfeitTokens` list, then i
            if (amounts[i] > 0 && LibHelpers.checkList(tokens[i]
                Collateral storage collatInfo = ts.collaterals[
                if (collatInfo.onlyWhitelisted > 0 && !LibWhitel
                    revert NotWhitelisted();
                if (collatInfo.isManaged > 0)
                    LibManager.release(tokens[i], to, amounts[i]
                else IERC20(tokens[i]).safeTransfer(to, amounts[
            }
            if (subCollateralsTracker[indexCollateral] - 1 <= i)
        }
        emit Redeemed(amount, tokens, amounts, forfeitTokens, ms
    }
```

During the collateral transfers(direct transfer or in `LibManager.release()`), there might be a hook for the recipient in the case of ERC777 tokens.

Then the recipient might charge more collaterals by reentrancy like this.

1. Let's suppose there are 2 collaterals `colA` and `colB`. The transmuter contract contains 1000 amounts of `colA` and `colB`. Alice has 20 amounts of agToken.
2. At the first time, Alice calls `redeem()` with 10 amounts of agToken and she should receive 10 amounts of `colA` and `colB`.
3. As `colA` is an ERC777 token, she calls `redeem(10)` again inside the hook after the **colA transfer**.
4. During the second redemption, total collaterals will be `colA = 990, colB = 1000` because `colB` isn't transferred in the first redemption yet.
5. After all, Alice will receive more collaterals in the second redemption from **this calculation**.

I think a similar reentrancy attack might be possible during the swap as well.

Recommended Mitigation Steps

I think we should add the `nonReentrant` modifier to the major functions like `redeem()/swap()`.

Picodes (Angle) confirmed, but disagreed with severity and commented:

> Valid. We had this in mind but thought it was ok as we don't plan to accept collaterals with callbacks. However better than sorry and we may add the modifier.

hansfriese (Judge) commented:

> @Picodes - `LibManager.release()` is called during the redemption and it might have a callback although the governance doesn't accept collaterals with hooks.

> Because the assumption is practical enough and the users can steal collaterals directly, will keep as High.

Angle mitigated:

> PR: https://github.com/AngleProtocol/angle-transmuter/commit/864c1c47cb550f8e337244f0f70409a171a4e671
> Adds a reentrancy guard to several functions.

Status: Mitigation confirmed. Full details in reports from auditor0517, Lambda, and Jeiwan.

[H-02] The first disputer might lose funds although his dispute is valid

Submitted by auditor0517, also found by Udsen and Jeiwan

Users can dispute the current tree using `disputeTree()` and the governor refunds the dispute funds if the dispute is valid in `resolveDispute()`.

```
function disputeTree(string memory reason) external {
    if (block.timestamp >= endOfDisputePeriod) revert Invali
    IERC20(disputeToken).safeTransferFrom(msg.sender, addres
    disputer = msg.sender;
    emit Disputed(reason);
}

/// @notice Resolve the ongoing dispute, if any
```

```
        /// @param valid Whether the dispute was valid
        function resolveDispute(bool valid) external onlyGovernorOr(
            if (disputer == address(0)) revert NoDispute();
            if (valid) {
                IERC20(disputeToken).safeTransfer(disputer, disputeA
                // If a dispute is valid, the contract falls back to
                _revokeTree();
            } else {
                IERC20(disputeToken).safeTransfer(msg.sender, disput
                endOfDisputePeriod = _endOfDisputePeriod(uint48(bloc
            }
            disputer = address(0);
            emit DisputeResolved(valid);
        }
```

But `disputeTree()` can be called again by another disputer although there is an active disputer and `resolveDispute()` refunds to the last disputer only.

In the worst case, a valid disputer might lose the dispute funds by malicious frontrunners.

1. A valid disputer creates a dispute using `disputeTree()`.
2. As it's valid, the governor calls `resolveDispute(valid = true)` to accept the dispute and refund the funds.
3. A malicious user calls `disputeTree()` by front running.
4. Then during `resolveDispute(true)`, the dispute funds will be sent to the second disputer and the first disputer will lose the funds although he's valid.

Recommended Mitigation Steps

`disputeTree()` shouldn't allow another dispute when there is an active dispute already.

Picodes (Angle) confirmed, but disagreed with severity and commented:

> Valid scenario and issue, although this is only a griefing attack, and the governance could still send back the funds to the first dispute using recoverERC20.

> Considering the scenario is very unlikely as it would cost gas to the attacker for nothing, and easily fixable, I think this should be downgraded to Med

hansfriese (Judge) commented:

> @Picodes - Will keep as High because honest disputers may lose their funds and it requires the governance's additional work to recover.

Picodes (Angle) commented:

> @hansfriese - indeed but when there is a dispute it requires additional work from the governance anyway. Like the permissions for `resolveDispute` and `recoverERC20`, and `recoverERC20` is even cheaper. So the trust assumptions of the disputer are exactly the same with and without this issue: he trusts the governance to send him back its deposit at some point.

> This whole `disputeAmount` / `period` thing is meant to prevent spam and to force disputers to behave correctly as they will lose some funds if they don't, so someone using this attack vector also exposes himself to the governance deciding to not accept the dispute and seize the funds. Overall we will of course respect your final decision but still think med is more appropriate here

hansfriese (Judge) commented:

> @Picodes - I totally understand your point and I'd like to mention two things.
>
> - The governance should refund one by one outside of the contract.
> - While checking `disputeTree()`, I see it doesn't emit the disputer's address and `disputeAmount` which would be changed later. So it wouldn't be that easy to refund in practice.
>
> I agree it's between High and Medium and will keep as High.

Angle mitigated:

> PR: https://github.com/AngleProtocol/merkl-contracts/commit/7402ee6b84789391479c5876b27be23fd579f7b2
> Applies the suggested fix.

Status: Mitigation confirmed. Full details in reports from Lambda, auditor0517, and Jeiwan.

[H-03] Poor detection of disputed trees allows claiming tokens from a disputed tree

Submitted by Jeiwan*, also found by* auditor0517

https://github.com/AngleProtocol/merkl-contracts/blob/1825925daef8b22d9d6c0a2bc7aab3309342e786/contracts/Distributor.sol#L200

Users can claim rewards from a Merkle tree that's being disputed. This can potentially lead to loss of funds since a malicious trusted EOA can claim funds from a malicious tree while it's being disputed.

Proof of Concept

The Distribution.getMerkleRoot function is used to get the current Merkle root during claiming. The function is aware of the dispute period of the current root and returns the previous root if the current tree is still in the dispute period.

However, the function doesn't take into account the situation when:

1. a tree was disputed (i.e. the disputer address is set);
2. and the dispute period has finished (i.e. when `block.timestamp >= endOfDisputePeriod`).

Such situations can happen realistically when a tree is disputed closer to the end of its dispute period and/or when the governor/guardian takes longer time to resolve the dispute. In such situations, the dispute period checks in the above functions will pass, however the `disputer` address will be set, which means that the tree is being disputed and shouldn't be used in claims.

As an example exploit scenario, a malicious trusted EOA can add a Merkle tree root that lets them claim the entire balance of the contract. Even if the tree gets disputed quickly, the success of the attack boils down to how quickly the governor/guardian

will resolve the dispute. To increase the chance, the attack can be deliberately executed when the governor/guardian are not active or available immediately.

Recommended Mitigation Steps

When the `disputer` address is set (after a call to `disputeTree`), consider treating the current tree as disputed, no matter whether the dispute period has passed or not. E.g. consider these changes:

```
diff --git a/contracts/Distributor.sol b/contracts/Distributor.s
index bc4e49f..8fb6a4c 100644
--- a/contracts/Distributor.sol
+++ b/contracts/Distributor.sol
@@ -197,7 +197,7 @@ contract Distributor is UUPSHelper {

     /// @notice Returns the MerkleRoot that is currently live f
     function getMerkleRoot() public view returns (bytes32) {
-        if (block.timestamp >= endOfDisputePeriod) return tree.
+        if (block.timestamp >= endOfDisputePeriod && disputer =
         else return lastTree.merkleRoot;
     }
```

Picodes (Angle) confirmed

Angle mitigated:

> PR: https://github.com/AngleProtocol/merkl-contracts/commit/82d8c0ff37b4a9ad8277cac4aef85f3ca0ad5c7c
> Applies the suggested fix.

Status: Mitigation confirmed. Full details in reports from Lambda, auditor0517, and Jeiwan.

Medium Risk Findings (7)

[M-01] LibHelpers.piecewiseLinear will revert when the value is less than the first element of the array

Submitted by auditor0517

https://github.com/AngleProtocol/angle-transmuter/blob/9707ee4ed3d221e02dcfcd2ebaa4b4d38d280936/contracts/transmuter/facets/Redeemer.sol#L156-L157
https://github.com/AngleProtocol/angle-transmuter/blob/9707ee4ed3d221e02dcfcd2ebaa4b4d38d280936/contracts/transmuter/libraries/LibSetters.sol#L230-L240
https://github.com/AngleProtocol/angle-transmuter/blob/9707ee4ed3d221e02dcfcd2ebaa4b4d38d280936/contracts/transmuter/libraries/LibHelpers.sol#L77-L80

`LibHelpers.piecewiseLinear` reverts when the value is less than the first element of the array. This method is used in Redeemer contract and if the collateral ratio is below the first element of xRedemptionCurve, the redepmtion will revert.

Proof of Concept

In `Redeemer._quoteRedemptionCurve`, a penalty factor is applied when the protocol is under-collateralized using `LibHelpers.piecewiseLinear`.

Redeemer.sol#L156-L157

```
            uint64[] memory xRedemptionCurveMem = ts.xRedemptionCurv
            penaltyFactor = uint64(LibHelpers.piecewiseLinear(collat
```

`xRedemptionCurveMem` is strictly increasing and upper bounded by `BASE_9`, and there's no more limitations.

LibSetters.sol

```
230         (action == ActionType.Redeem && (xFee[n - 1] > BASE_9

233         for (uint256 i = 0; i < n - 1; ++i) {
234             if (

240                 (action == ActionType.Redeem && (xFee[i] >= x
```

So `collatRatio` can be less than the first element of `xRedemptionCurveMem`. In that case, `LibHelpers.findLowerBound` will return 0 and `LibHelpers.piecewiseLinear` will revert on the following line.

LibHelpers.sol#L77-L80

```
        return
            yArray[indexLowerBound] +
            ((yArray[indexLowerBound + 1] - yArray[indexLowerBound])
            int64(xArray[indexLowerBound + 1] - xArray[indexLowerBou
```

`Redeemer._redeem` calls `_quoteRedemptionCurve`, so the redemption will be blocked in this case.

Recommended Mitigation Steps

We can add the following line to mitigate this issue.

```
        if (indexLowerBound == 0 && x < xArray[0]) return yArray[0];
```

Picodes (Angle) confirmed and commented:

> Indeed we are assuming throughout the test base that the first value is 0 but the check is missing in the code so there this could happen. It's kind of conditional to a misconfiguration though. Leaving it up to the judge.

hansfriese (Judge) commented:

> @Picodes - Medium is appropriate because it's likely to happen with the current configuration.

Angle mitigated:

> PR: https://github.com/AngleProtocol/angle-transmuter/commit/5f7635cdab52b75416309d45f8cd253609c705ff
> Add a handler for this edge case.

Status: Mitigation confirmed. Full details in reports from Lambda, auditor0517, and Jeiwan.

[M-02] Unsafe cast in `getCollateralRatio()`

Submitted by auditor0517, also found by Lambda

`LibGetters.getCollateralRatio()` might return the incorrect ratio due to the unsafe cast.

Proof of Concept

`getCollateralRatio()` outputs the collateral ratio using the total collaterals and issued agTokens.

```
        // The `stablecoinsIssued` value need to be rounded up becau
        // the amount of stablecoins issued
        stablecoinsIssued = uint256(ts.normalizedStables).mulDiv(ts.
        if (stablecoinsIssued > 0)
            collatRatio = uint64(totalCollateralization.mulDiv(BASE_
        else collatRatio = type(uint64).max;
```

Typically, the `collatRatio` should be around `BASE_9` but the ratio might be larger than `type(uint64).max` during the initial stage.

Furthermore, `totalCollateralization` is calculated using the raw balance of collaterals and it might be manipulated when stablecoinsIssued is not large.

Then collatRatio might be cast to the wrong value.

After all, `getCollateralRatio()` will return the wrong ratio and it will affect the protocol seriously.

Recommended Mitigation Steps

I think we should use the SafeCast library in getCollateralRatio().

Picodes (Angle) disagreed with severity and commented:

> This seems hardly doable. Assuming there is 1 stablecoin issued (1e18), you'd need 1e28 collateral. Furthermore no impact is described: this would just break this view function. Overall I think Low severity is more appropriate

hansfriese (Judge) commented:

> @Picodes - Although it's an edge case, it's likely to happen and `getCollateralRatio()` plays an important role in the protocol. Will keep as Medium.

GuillaumeNervoXS (Angle) commented:

> To overflow the protocol would need to be 1 billion % over collateralise so it is not likely to happen.

hansfriese (Judge) commented:

> As #9 shows, 1 USD is enough when `stablecoinsIssued = 1 wei`. I think it should be mitigated for safety.

Angle mitigated:

> PR: https://github.com/AngleProtocol/angle-transmuter/commit/6f2ffcb1e89e3bba05c9aa2133ef94347aa42c28
> Adds safeCast.

Status: Mitigation error. Full details in reports from Lambda, auditor0517, and Jeiwan - and also shared below in the Mitigation Review section.

[M-03] Read-only reentrancy is possible

Submitted by auditor0517

https://github.com/AngleProtocol/angle-transmuter/blob/9707ee4ed3d221e02dcfcd2ebaa4b4d38d280936/contracts/tra

nsmuter/facets/Swapper.sol#L206
https://github.com/AngleProtocol/angle-transmuter/blob/9707ee4ed3d221e02dcfcd2ebaa4b4d38d280936/contracts/transmuter/facets/Redeemer.sol#L131
https://github.com/AngleProtocol/angle-transmuter/blob/9707ee4ed3d221e02dcfcd2ebaa4b4d38d280936/contracts/savings/SavingsVest.sol#L110

The agToken might be minted wrongly as rewards due to the reentrancy attack.

Proof of Concept

There are `redeem/swap` logics in the `transmuter` contract and all functions don't have a `nonReentrant` modifier.

So the typical reentrancy attack is possible during `redeem/swap` as I mentioned in my other report.

But besides that, the read-only reentrancy attack is possible from the `SavingsVest` contract, and the agToken might be minted/burnt incorrectly like this.

1. The collatRatio is `BASE_9(100%)` now and Alice starts a swap from collateral to agToken in `Swapper` contract.
2. In `_swap()`, it mints the agToken after depositing the collaterals.

```
if (mint) {
    uint128 changeAmount = (amountOut.mulDiv(BASE_27, ts.nor
    // The amount of stablecoins issued from a collateral ar
    // as variables normalized by a `normalizer`
    collatInfo.normalizedStables += uint216(changeAmount);
    ts.normalizedStables += changeAmount;
    if (permitData.length > 0) {
        PERMIT_2.functionCall(permitData);
    } else if (collatInfo.isManaged > 0)
        IERC20(tokenIn).safeTransferFrom(
            msg.sender,
            LibManager.transferRecipient(collatInfo.managerI
            amountIn
        );
    else IERC20(tokenIn).safeTransferFrom(msg.sender, addres
```

```
            if (collatInfo.isManaged > 0) {
                LibManager.invest(amountIn, collatInfo.managerData.c
            }
            IAgToken(tokenOut).mint(to, amountOut);
        }
```

After depositing the collaterals, Alice might have a hook in the case of ERC777 tokens before the agToken is minted.

3. Then Alice calls `SavingsVest.accrue()` inside the hook and getCollateralRatio() will return the incorrect ratio as the agToken isn't minted yet.
4. So `collatRatio` will be larger than the real value and additional rewards would be minted if `collatRatio > BASE_9 + BASE_6`.

Then Alice would get more rewards from the `SavingsVest`.

Recommended Mitigation Steps

Recommend adding the `nonReentrant` modifer to getCollateralRatio() as well as `redeem()/swap()` functions.

Picodes (Angle) confirmed and commented:

> So the assumption is that there is an accepted collateral `ERC777`, which is really unlikely as there is no credible candidate and it would bring additional risk. But the scenario is valid.

Angle mitigated:

> PR: https://github.com/AngleProtocol/angle-transmuter/commit/864c1c47cb550f8e337244f0f70409a171a4e671
> Adds a reentrancy guard to several functions.

Status: Mitigation confirmed. Full details in reports from Lambda, auditor0517, and Jeiwan.

[M-04] `estimatedAPR()` might return the wrong APR

Submitted by auditor0517

`estimatedAPR()` might return the wrong APR and it will make users confused.

Proof of Concept

`SavingsVest.estimatedAPR()` returns the APR using the current `vestingProfit` and `vestingPeriod`.

```
        function estimatedAPR() external view returns (uint256 apr)
            uint256 currentlyVestingProfit = vestingProfit;
            uint256 weightedAssets = vestingPeriod * totalAssets();
            if (currentlyVestingProfit != 0 && weightedAssets != 0)
                apr = (currentlyVestingProfit * 3600 * 24 * 365 * BA
        }
```

First of all, it uses the current `vestingRatio = vestingProfit / vestingPeriod` for 1 year even if `vestingPeriod < 1 year`. I think it might be an intended behavior to estimate the APR with the current vesting ratio.

But it's wrong to use the same vesting ratio after the vesting period is finished already.

In accrue(), it updates the `vestingProfit` and `lastUpdate` only when it's overcollateralized/undercollateralized more than 0.1%.

So `lastUpdate` wouldn't be changed for a certain time while collatRatio is in range (99.9%, 100.1%).

1. At the first time, `vestingProfit = 100, vestingPeriod = 10 days` and `estimatedAPR()` returns the correct value.
2. After 10 days, all vestings are unlocked and there is no locked profit. But `accrue()` has never been called due to the stable collateral ratio.
3. In `estimatedAPR()`, `vestingProfit` will be 100 and it will return the same APR as 10 days before.

4. But the APR should be 0 as there is no locked profit now.

Recommended Mitigation Steps

`estimatedAPR()` should return 0 when `lockedProfit() == 0`.

```
        function estimatedAPR() external view returns (uint256 apr)
            if (lockedProfit() == 0) return 0; //check locked profit

            uint256 currentlyVestingProfit = vestingProfit;
            uint256 weightedAssets = vestingPeriod * totalAssets();
            if (currentlyVestingProfit != 0 && weightedAssets != 0)
                apr = (currentlyVestingProfit * 3600 * 24 * 365 * BA
        }
```

Picodes (Angle) confirmed and commented:

> Valid, although this function isn't really useful as what matters in the end is how much a call to `accrue` would give. `estimatedAPR` should return an approximation of the current APR but is an approximation on the long run

hansfriese (Judge) commented:

> @Picodes - It looks like a middle of Medium and Low as it's a view function. Will keep as Medium because it will be used to estimate the profit rate for users.

Angle mitigated:

> PR: https://github.com/AngleProtocol/angle-transmuter/commit/337c65d005bbd8ed6dfa76929d2cae475066756a
> Applies the suggested fix.

Status: Mitigation confirmed. Full details in reports from auditor0517 and Jeiwan.

[M-05] uint128 `changeAmount` might overflow

Submitted by __141345__

https://github.com/AngleProtocol/angle-transmuter/blob/9707ee4ed3d221e02dcfcd2ebaa4b4d38d280936/contracts/transmuter/facets/Swapper.sol#L189
https://github.com/AngleProtocol/angle-transmuter/blob/9707ee4ed3d221e02dcfcd2ebaa4b4d38d280936/contracts/transmuter/facets/Swapper.sol#L210

This issue is an edge case, that uint128 `changeAmount` could overflow, making the protocol fail for certain amount of swap.

Proof of Concept

Let's break down the `changeAmount` :

1. `amountOut/amountIn`
2. BASE_27
3. `normalizer`

```
File: transmuter/contracts/transmuter/facets/Swapper.sol
176:     function _swap() {

189:                 uint128 changeAmount = (amountOut.mulDiv(BA

210:                 uint128 changeAmount = ((amountIn * BASE_27
```

1. `normalizer`

could take value ranging from 1e18 to 1e36, we take the worst case, 1e18.

```
File: transmuter/contracts/transmuter/facets/Redeemer.sol
193:         if (newNormalizerValue <= BASE_18 || newNormalizerV

208:             newNormalizerValue = BASE_27;
209:         }
```

2. BASE_27 is constant 1e27.
3. `amountOut/amountIn` is AgToken with 18 decimals.

Combine the above 1, 2, 3, we have the “total decimal”: 1e18 * 1e27 / 1e18 = 1e27.

The max value of uint128 is 3.4e38. To overflow, the `amountOut/amountIn` need to be around 3.4e38 / 1e27 = 3.4e11.

For nowadays currencies around the world, we can look at Iranian Rial (IRR). 3.4e11 IRR is around 8M USD. 8M USD is a big amount for now. However considering the following, this concern maybe not nonsense:

- USD itself is depreciating, from 1970s to now, the gold price(historical chart) has changed from 35 USD/oz to 2000 USD/oz, almost 60 times.
- Some countries have high inflation rate (Iran, Vietnam, Sierra, Leonean, Lao, etc. That’s how the many zeros in these currencies come).
- Taking a longer time span of 10-20 years, the depreciation could be influential.

If after 20 years, 3.4e11 of some currency with high inflation depreciates more than 100 times . In that case, the amount is around 80k USD, a reasonable amount to use (especially for institutions).

Although the possibility is low for current use cases, in the future, things could change. To make the stablecoin protocol universal for all countries in the world, the edge case might be some concern.

Recommended Mitigation Steps

- Use uint256 for `changeAmount` , then the likelihood would be negligible small, making the system more robust to special situations. uint256 wont add too much cost, but can avoid much more potential issue.
- Use narrower range for `normalizer` . 1e9 for both up and down direction might be too big, and not necessary.

Picodes (Angle) disagreed with severity and commented:

> Although the possibility of overflow is real, I don’t think the scenario is convincing and this likely to happen. Worst case the code is already using safeCast.

hansfriese (Judge) commented:

> @Picodes - Will keep it as medium because it's likely to happen and `_swap()` will revert during the safeCast.

Angle chose not to mitigate:

> We consider that there is no risk here as swaps will be reverting, and that the chances that this happen are infinitesimals.

[M-06] Interest is not accrued before parameters are updated in `SavingsVest`

Submitted by **Jeiwan**

Stablecoin holders can receive wrongly calculated yield in the `SavingsVest` contract. Also, wrong vesting profit can be slashed when the protocol is under-collateralized.

Proof of Concept

The **SavingsVest** contract lets users deposit their stablecoins and earn vested yield when the stablecoin in the Transmuter protocol is over-collateralized. The interest is accrued via calls to the **SavingsVest.accrue** function.

There are two parameters that affect the profit of depositors:

1. **protocolSafetyFee** is the fees paid to the protocol;
2. **vestingPeriod** is the period when the yield remains locked.

The two parameters can be changed via the **setParams** function. However, before they're changed, the current interest is not accrued. E.g. this may lead to:

1. If `protocolSafetyFee` is increased without accruing interest, the next accrual will happen at the increased fees, which will reduce the rewards for the depositors.
2. If `vestingPeriod` is increased without accruing interest, the yield will be **locked for a longer period** and the next accrual may **slash more vested yield**.

Thus, users can lose a portion of the yield that was earned at a lower protocol fee after the fee was increased. Likewise, increasing the vesting period may result in slashing yield that was earned before the period was increased.

Recommended Mitigation Steps

In the `SavingsVest.setParams` function, consider accruing interest with the current parameters before setting new `protocolSafetyFee` and `vestingPeriod`.

Picodes (Angle) confirmed and commented:

> Confirmed. The worst scenario is that when modifying the `vestingPeriod` it can lead to a jump in `lockedProfit`, so to a jump in `totalAssets()`. So eventually someone could sandwich attack the update for a profit.

Angle mitigated:

> PR: https://github.com/AngleProtocol/angle-transmuter/commit/94c4e51ae3400a63532e85f04f4081152adc97db
> Calls `accrues` before updating sensible parameters.

Status: Mitigation confirmed. Full details in report from Jeiwan.

[M-07] User may get less tokens than expected when collateral list order changes

Submitted by Lambda

https://github.com/AngleProtocol/angle-transmuter/blob/8a2c3aaf4bd054581b06d33049370a6f01b56d44/contracts/transmuter/libraries/LibSetters.sol#L123
https://github.com/AngleProtocol/angle-transmuter/blob/8a2c3aaf4bd054581b06d33049370a6f01b56d44/contracts/transmuter/facets/Redeemer.sol#L64

The order of `ts.collateralList` is not stable: Whenever `LibSetters.revokeCollateral` is used to revoke a collateral, it may change

because of the swap that is performed. However, the function `Redeemer.redeem` relies on this order, as the user has to provide the `minAmountsOut` in the order of `ts.collateralList`. This can lead to situations where the user has crafted the `minAmountsOut` array when the order was still different, leading to unintended results (and potentially redemptions that the user did not want to accept). It also means that revoking a collateral can be challenging for the team / governance because it should never be done when a user has already prepared a redemption (either via the frontend which he had open or some other way to interact with the contract). But there is of course no way to know this.

Proof of Concept

Let's say the system contains the collateral [tokenA, tokenB, tokenC]. `normalizedStables` for tokenA is 0. The user therefore does not want to receive tokenA (and will not receive anything for it). However, it is extremely important to him that he receives 100,000 of tokenC. He therefore crafts a `minAmountsOut` of [0, 10000, 100000]. Just before he submits the call, tokenA is removed from the system, resulting in the collateral array [tokenC, tokenB]. Even if the user only receives 50,000 tokens of tokenC, the call will therefore succeed.

Recommended Mitigation Steps

The problem could be alleviated a bit by checking the length of `minAmountsOut` (making sure it is not longer than `ts.collateralList`). However, that would not help if a collateral is revoked and a new one is added. Another solution would be to provide pairs of token addresses and amounts, which would solve the problem completely.

Picodes (Angle) confirmed and commented:

> The mitigation doesn't cost much and we will implement it. It's really an edge case though.

Angle mitigated:

> PR: https://github.com/AngleProtocol/angle-transmuter/commit/f8d0bf7c4009586f7022d5929359041db3990175
> Applies the suggested fix.

Status: Not fully mitigated. Full details in reports from Lambda and auditor0517.

Low Risk and Non-Critical Issues

For this audit, 4 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Lambda** received the top score from the judge.

The following wardens also submitted reports: Udsen, auditor0517, *and* Jeiwan .

[L-01] `DiamondProxy` License

`DiamondProxy` is a fork of https://github.com/mudgen/diamond-3/blob/master/contracts/Diamond.sol, but has changed the license from MIT to CC0-1.0. While this should be no problem legally (as far as I know), there is no reason to change the license in my opinion.

[L-02] Nested Loop in `Redeemer._redeem`

The function `_redeem` iterates over all tokens. Moreover, the check `LibHelpers.checkList(tokens[i], forfeitTokens)` (which is linear in the length of `forfeitTokens`) is performed within loop iteration. Therefore if there are n tokens and m tokens to be forfeited, the overall complexity will be $O(n*m)$. This can be very expensive depending on the values of n and m. As an alternative, the forfeited tokens could be stored in a mapping in the beginning and this mapping could be queried (constant complexity) within each iteration.

[L-03] (Temporary) manipulations of `IManager.totalAssets()` would break the system

One potential attack vector for the system is (temporarily) manipulating the return value of `IManager.totalAssets()`. Depending on how the strategies are implemented, it may be possible to inflate their values, for instance by using flash loans and temporarily depositing / withdrawing. Because no strategies were in-scope, it is not possible to judge if this is currently possible, but it is something to keep in mind (and potentially document) when building strategies.

[L-04] Third Order Taylor Approximation can be imprecise

The `Savings` contract uses a third-order taylor approximation for approximating $(1 + i)^n$ (i.e., continuous compounding). However, this can be imprecise for large value of ni. For instance, consider the difference when the assets are 10,000, $i = 0.01$, $n = 10$:

```
10000*(1.01)^{10} - 10000*(1+0.01*10+10*9*0.01^2/2+10*9*8*0.01^3
```

Now, consider the difference for $i = 0.000001$ and $n=864000$ (one week when n denotes seconds):

```
10000*(1 + 0.000001)^{864000} - 10000*(1+0.000001*864000+864000*
```

An alternative would be to use a library (which usually contains higher-order terms) for the exponentiation.

[L-05] Grifting possibilities for strategies

`LibSetters.revokeCollateral` performs the following check for managed collateral:

```
(, uint256 totalValue) = LibManager.totalAssets(collatInfo.manag
if (totalValue > 0) revert ManagerHasAssets();
```

Depending on the strategy used, there may be ways where another user can increase `totalAssets` (e.g., donating 1 wei), making the removal of a collateral impossible.

[L-06] Collateral may not be revokable because of rounding

`LibSetters.revokeCollateral` performs the following check:

```
if (collatInfo.decimals == 0 || collatInfo.normalizedStables > (
```

However, because of the rounding within `_swap`, it can happen that `normalizedStables` is a very small value (e.g., 1 wei) and it is not possible to decrease it further.

[L-07] Swapper assumes 18 decimals for the stable coin

The `Swapper` contract assumes in multiple places that the stable coin contract has 18 decimals (e.g., here). While this is true for `AgToken`'s, the system is intended to be general and usable with other underlying stablecoins according to the whitepaper. Therefore, making this configurable would make sense in my opinion.

[N-01] `RewardHandler.sellRewards` requires trust in seller

There is a check within `RewardHandler.sellRewards` to confirm that the balance of collateral assets has increased. However, this check does not change the fact that trust in the seller is required. A malicious seller could still craft an order with huge slippage that sells tokens at a very poor rate and only increases collateral slightly. An alternative would be to construct the 1inch payload dynamically instead of relying on the user to provide it.

[N-02] Reliance on `decimals()`

`LibSetters.addCollateral` queries `decimals()` and therefore relies on the fact that tokens implement this function. While this is not guaranteed and generally against EIP 20 ("OPTIONAL - This method can be used to improve usability, but interfaces and other contracts MUST NOT expect these values to be present."), it should not be a huge problem in practice because most ERC20 tokens implement the functions. Nevertheless, because the collateral is provided by the user anyways, an (optional) alternative would be to let him also provide the decimals.

[N-03] System may use prices that never existed

Becuase `LibOracle.read` combines multiple Chainlink feeds with a different timestamp, it may use a price for an asset that has never existed (when it multiplies prices with timestamp X and Y). This has for instance happened to Y2K in the past. It is quite hard to avoid (one could keep multiple prices and try to find a "closest match" based on the history, but this is also not perfect), but something to keep in mind.

Status: See [Mitigation Review] section below for details regarding mitigations related to low and non-critical issues.

Gas Optimizations

For this audit, 2 reports were submitted by wardens detailing gas optimizations. The [report highlighted below] by **Udsen** received the top score from the judge.

The following warden also submitted a report: [auditor0517].

[G-01] Unused variables can be omitted

In the `LibOracle.getBurnOracle()` function, declares the `uint256 oracleValueTmp` memory variable and later initializes it as follows by calling the `readBurn` function:

```
(oracleValueTmp, ratioObserved) = readBurn(ts.collaterals[cc
```

But the `oracleValueTmp` variable is never used within the scope of the function and is used only as a place holder. Hene we can omit this variable and save extra `MSTORE` for each iteration of the `for` loop. Hence the gas saved will be equal to `3 * collateralList.length` ; As a result if the list of collateral assets supported by this system is higher this will save considerable amount of gas.

The line of code can be edited as follows:

```
( , ratioObserved) = readBurn(ts.collaterals[collateralList|
```

https://github.com/AngleProtocol/angle-transmuter/blob/9707ee4ed3d221e02dcfcd2ebaa4b4d38d280936/contracts/transmuter/libraries/LibOracle.sol#L79-L84

[G-02] `storage` variable can be replaced with `calldata` variable

In the `Swapper._buildPermitTransferPayload` the `Collateral` struct is passed into the function as a `storage` variable named `collatInfo`. But it is only read from and not written to within the scope of the function implementation. Hence it is recommended to make the `collatInfo` a `calldata` variable and read from `calldata` instead, to save on two extra `SLOAD` operations for the transaction.

https://github.com/AngleProtocol/angle-transmuter/blob/9707ee4ed3d221e02dcfcd2ebaa4b4d38d280936/contracts/transmuter/facets/Swapper.sol#L497
https://github.com/AngleProtocol/angle-transmuter/blob/9707ee4ed3d221e02dcfcd2ebaa4b4d38d280936/contracts/transmuter/facets/Swapper.sol#L500

[G-03] Redundant checks can be omitted

In the `Swapper.swapExactOutputWithPermit()` function performs a check to verify that the `amountIn` is less than the `amountInMax` amount as follows:

```
if (amountIn > amountInMax) revert TooBigAmountIn();
```

But this is a redundant check since the same check is performed as shown below, in the `SignatureTransfer.sol` contract which is a parent contract of `Permit2.sol` contract. This is called inside the `Swapper._swap()` function.

```
if (requestedAmount > permit.permitted.amount) revert Invali
```

Hence the redundant call inside the `Swapper.swapExactOutputWithPermit()` function can be omitted to save gas.

https://github.com/AngleProtocol/angle-transmuter/blob/9707ee4ed3d221e02dcfcd2ebaa4b4d38d280936/contracts/transmuter/facets/Swapper.sol#L139

[G-04] Conditional check can be performed during variable initialization

In the `DistributionCreator._createDistribution()` function the `userFeeRebate` value is used when calculating the `distributionAmountMinusFees` if the `UniswapV3Pool` tokens are not whitelisted. Before the `distributionAmountMinusFees` calculatation, there is a check inside the `if` statement as follows:

```
userFeeRebate < BASE_9
```

This check makes sure that the `distributionAmountMinusFees` will only be calculated if the `userFeeRebate < BASE_9`.

But it is recommended to check whether `userFeeRebate < BASE_9` in the `DistributionCreator.setUserFeeRebate` function when setting the `feeRebate[user] = userFeeRebate`. By doing this we can omit the `userFeeRebate < BASE_9` check in the `DistributionCreator._createDistribution()` and also we can prevent setting the `feeRebate[user]` value as invalid (> BASE_9) value thus saving a `SSTORE` as well.

The suggested code snippet change is as follows:

```
function setUserFeeRebate(address user, uint256 userFeeRebate) e
    require(userFeeRebate < BASE_9, "Invalid userFeeRebate");
    feeRebate[user] = userFeeRebate;
    emit FeeRebateUpdated(user, userFeeRebate);
}
```

https://github.com/AngleProtocol/merkl-contracts/blob/1825925daef8b22d9d6c0a2bc7aab3309342e786/contracts/DistributionCreator.sol#L392-L395
https://github.com/AngleProtocol/merkl-contracts/blob/1825925daef8b22d9d6c0a2bc7aab3309342e786/contracts/DistributionCreator.sol#L240

[G-05] `uint256` variable initialization to default value of `0` can be omitted

There is no need to initialize variables to their default values during declaration, since they are any way initialized to default value once declared.

https://github.com/AngleProtocol/angle-transmuter/blob/9707ee4ed3d221e02dcfcd2ebaa4b4d38d280936/contracts/transmuter/facets/Redeemer.sol#L199

[G-06] Arithmatic operations can be unchecked, since there is no underflow or overflow

In the `DistributionCreator.toggleTokenWhitelist` function implementation can be `unchecked` to save gas.

This function is used to toggle the value of `isWhitelistedToken[token]` between `0` adn `1`. Hence the arithematic operation can never underflow or overflow. Hence the `DistributionCreator.toggleTokenWhitelist` function can be updated as follows with the `unchecked` keyword.

```
function toggleTokenWhitelist(address token) external onlyGoverr
    unchecked{
        uint256 toggleStatus = 1 - isWhitelistedToken[token];
        isWhitelistedToken[token] = toggleStatus;
    }
    emit TokenWhitelistToggled(token, toggleStatus);
}
```

https://github.com/AngleProtocol/merkl-contracts/blob/1825925daef8b22d9d6c0a2bc7aab3309342e786/contracts/DistributionCreator.sol#L398-L402
https://github.com/AngleProtocol/merkl-contracts/blob/1825925daef8b22d9d6c0a2bc7aab3309342e786/contracts/Distributor.sol#L266-L267
https://github.com/AngleProtocol/merkl-contracts/blob/1825925daef8b22d9d6c0a2bc7aab3309342e786/contracts/Distr

ibutor.sol#L273-L274
https://github.com/AngleProtocol/merkl-contracts/blob/1825925daef8b22d9d6c0a2bc7aab3309342e786/contracts/Distributor.sol#L208-L209
https://github.com/AngleProtocol/angle-transmuter/blob/9707ee4ed3d221e02dcfcd2ebaa4b4d38d280936/contracts/savings/Savings.sol#L221-L223

In the `SavingsVest.accrue()` function, when the protocol is under-collateralized and the `missing <= currentLockedProfit` the vesting profit updated as follows:

```
} else {
    vestingProfit = currentLockedProfit - missing;
    lastUpdate = uint64(block.timestamp);
}
```

This arithmetic operation can be unchecked due to previous check of `if (missing > currentLockedProfit)`.

Hence the above code snippet can be modified as follows:

```
} else {
    unchecked{
        vestingProfit = currentLockedProfit - missing;
        lastUpdate = uint64(block.timestamp);
    }
}
```

https://github.com/AngleProtocol/angle-transmuter/blob/9707ee4ed3d221e02dcfcd2ebaa4b4d38d280936/contracts/savings/SavingsVest.sol#L134-L137

Status: See Mitigation Review section below for details regarding mitigations related to gas optimizations.

Audit Analysis

For this audit, 2 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The **report highlighted below** by **__141345__** received the top score from the judge.

The following wardens also submitted reports: **auditor0517** *and* **Lambda**.

Composability

Composability is a huge advantage of DeFi, on the other side, it also could be a double sword. Considering Angle can work with other stablecoin system, and the sub-collateral is also introduced, the inter dependency could go quite complex later on. Some troublesome situation could be like: stableA use stableB as collateral, stableB use stableC as collateral, stableC again use stableA as collateral, ABC form a cycled collateral. If the cycle involves tens of tokens, it could be hard to spot the issue. But in the time of crisis, such inter dependent system will break quickly.

The idea of sub-collateral can help to expand the available assets for the protocol. On the other hand, more attention will be needed with regards to uncontrollable external contracts. It may need special attention, such as:

- the sub-collateral changed the constitution (such as DAI and FRAX could change the backed assets).
- it may further has sub-sub-collateral.
- there is locked time period of the sub-collateral, such as notional fixed rate bond token, maybe 6 month to expiry.

The same principal applies to other parts of the contract. Overall, the protocol works great in terms of modular implementation, by which I mean the Angle system has sub-modules, acting like LEGO. One developing suggestion might be not go too far in the compatibility with other protocols, just like the sub-collateral, maybe one layer of integration is good enough, but not too much.

Penalty Factor and Depeg Handling

The main suggestion is to add some kind of downside protection as last resort. Below is the reason.

The penalty factor can deter users from bank run, this is a clever design. But depending on the severity of the situation, the outcome might be quite different. Let's divide the users into 2 groups: "run" and "stay", and compare the 2 past big events: USDC and luna.

1. USDC
   It depegged to 0.87 and recovered afterwards. The penalty rule will work great. The "run" group will be penalized and the "stay" group will be rewarded. The most important is, USDC recovered at last.
2. luna
   It just went down to zero. At the end, everyone loses. At the beginning, the "run" group would be discouraged to exit. In the end, they ended up losing more because missing the earlier exit stop loss opportunity.

My understanding of some main stream stable coin is, it should stay pegged, or just go zero and exit. It's hard to imagine USDC valued at $0.65 after market rally (even if the loss is too big, the admin might try to burn some of the supply to recover the price, otherwise it has nothing to do with USD). Hence, the penalty factor rule can work great in the first situation with recover, but not for collapse case. When depeg comes, the protocol will hope no one get panic and bank run, but can not assure that everything is ok if staying. It will become a difficult trade off for users.

To deal with the worst situation, some protect such as insurance can be added. With the downside protection as last resort, users wont worry that much to join the "stay" group. Worst case, if this stable coin goes to zero, they know they are still covered anyway. The last protection could act as a patch for the penalty rule.

There are some defi insurance providers, for example unslashed, nexusmutual, insurace, bridgemutual. As for the choice of this kind of protection, the cheapest tier can be used, since the collapse situation is some extremely low possible black swan. The alternative could be put option, however the choice could be limited, and the cost might be higher.

One more thing, for such kind of insurance, if try to protect USDC, the payout should not be in USDC. Because if USDC goes to 0, it it pointless to receive huge payout still in USDC.

Mitigation Review

Introduction

Following the C4 audit, 3 wardens (Lambda, auditor0517, and Jeiwan) reviewed the mitigations for all identified issues. Additional details can be found within the C4 Angle Protocol Mitigation Review repository.

Overview of Changes

Summary from the Sponsor:

> Changes related to High and Medium issues on Merkl can be found here.

> Changes that we intend to make on Merkl prior to final deployment, so including QA and GAS can be found here. Updated tests and scripts are on the `main` branch.

> Changes related to High and Medium issues on Transmuter can be found here.

> Changes that we intend to make on Transmuter prior to final deployment, so including QA and GAS can be found can be found here. Updated tests and scripts are on the `main` branch.

Mitigation Review Scope

URL	Mitigation of	Purpose
https://github.com/AngleProtocol/angle-transmuter/commit/864c1c47cb550f8e337244f0f70409a171a4e671	H-01	Adds a reentrancy guard to several functions
https://github.com/AngleProtocol/merkl-contracts/commit/7402ee6b84789391479c5876b27be23fd579f7b2	H-02	Applies the suggested fix
https://github.com/AngleProtocol/merkl-contracts/commit/82d8c0ff37b4a9ad8277cac4aef85f3ca0ad5c7c	H-03	Applies the suggested fix
https://github.com/AngleProtocol/angle-transmuter/commit/5f7635cdab52b75416309d45f8cd	M-01	Add a handler for this edge case

URL	Mitigation of	Purpose
253609c705ff		
https://github.com/AngleProtocol/angle-transmuter/commit/6f2ffcb1e89e3bba05c9aa2133ef94347aa42c28	M-02	Adds safeCast
https://github.com/AngleProtocol/angle-transmuter/commit/864c1c47cb550f8e337244f0f70409a171a4e671	M-03	Adds a reentrancy guard to several functions
https://github.com/AngleProtocol/angle-transmuter/commit/337c65d005bbd8ed6dfa76929d2cae475066756a	M-04	Applies the suggested fix
https://github.com/AngleProtocol/angle-transmuter/commit/94c4e51ae3400a63532e85f04f4081152adc97db	M-06	Calls `accrues` before updating sensible parameters
https://github.com/AngleProtocol/angle-transmuter/commit/f8d0bf7c4009586f7022d5929359041db3990175	M-07	Applies the suggested fix
https://github.com/AngleProtocol/merkl-contracts/commit/3c2fe3a956cdd29b632e8d7a20e1fc2ce5e8ac37	QA & GAS	
https://github.com/AngleProtocol/angle-transmuter/commit/66bba3f5dba4ab6307c997e350dfadb13d2a2119	QA & GAS	

Mitigation Review Summary

Original Issue	Status	Full Details
H-01	Mitigation Confirmed	Reports from auditor0517, Lambda, and Jeiwan
H-02	Mitigation Confirmed	Reports from Lambda, auditor0517, and Jeiwan
H-03	Mitigation Confirmed	Reports from Lambda, auditor0517, and Jeiwan
M-01	Mitigation Confirmed	Reports from Lambda, auditor0517, and Jeiwan
M-02	Mitigation Error	Reports from Lambda, auditor0517, and Jeiwan
M-03	Mitigation Confirmed	Reports from Lambda, auditor0517, and Jeiwan
M-04	Mitigation Confirmed	Reports from auditor0517 and Jeiwan
M-06	Mitigation Confirmed	Report from Jeiwan
M-07	Not Fully Mitigated	Reports from Lambda and auditor0517

Original Issue	Status	Full Details
QA	Mitigation Confirmed	Reports from auditor0517 and Lambda
Gas	Mitigation Confirmed	Reports from auditor0517 and Lambda

The wardens determined that one Medium severity finding was not fully mitigated. They also surfaced one mitigation error of Medium severity. See below for full details on both.

M-02 Mitigation Error: Attacker can brick redemptions by donating a small amount

Submitted by Lambda*, also found by* auditor0517 *and* Jeiwan

While the fix properly fixes the issue of collateralization ratio overflows (that can no longer occurs), it enables DoS attacks on the redemption mechanism:

Issue description

Consider the example that was already provided (https://github.com/code-423n4/2023-06-angle-findings/issues/9), where an attacker produces an overflow by donating a small amount:

> Let's take an extreme example and say that `stablecoinsIssued` is 1 (although the attack also works with larger values). If a user now transfer 1 USD of a stablecoin to the contract, `totalCollateralization` will be roughly `10**18`. Therefore, the system calculates:

```
collatRatio = uint64(10**18 * 10**9 / 1) = uint64(10**27);
```

> Because `10**27 > type(uint64).max`, this overflows.

After the fix, any call to `getCollateralRatio` will revert because of the `SafeCast`. Therefore, any call to `_quoteRedemptionCurve` will also revert, meaning that redemptions fail.

Note that the attacker never directly interacted with the contract (as he donated the tokens), which means that the protection also does not work.

Recommendation

It is recommended to set `collatRatio` to `type(uint64).max` instead of reverting when an overflow happens. Then, the problem would no longer occur.

sogipec (Angle) commented:

> This is indeed a valid remark. That being said, I don't think we'll adjust the code for this one, and we still prefer the function to revert if the collateral ratio overflows, as we hardly see a case where anyone would redeem in such a situation (gas cost on Ethereum far greater than the amount that can be redeemed given the low amount of stablecoinsIssued for it to happen). We'll try to put a comment to encourage initialization of Transmuter with a non null `stablecoinsIssued` amount (like >1, but closer to `10€` worth of stablecoins) so this makes this DoS attack economically impossible.

M-07 Unmitigated

Submitted by **Lambda**, *also found by* **auditor0517**

The fix addresses the scenarios when collaterals are removed between the crafting of the `minAmountsOut` list and the submission of the transaction. Then, we will have `amounts.length > minAmountOuts.length`, meaning that the following line causes a revert:

```
if (amountsLength != minAmountOuts.length) revert Invali
```

However, as mentioned in the issue, only checking the length does not mitigate the issue fully. If one collateral is removed and another one added, it still exists. For instance, we could have:

- Initial collateral list [A, B, C, D]. This is used by the user to craft `minAmountOuts`
- Then, collateral A is removed, resulting in [D, B, C].

- Then, a new collateral Z is added, resulting in [D, B, C, Z].

We have `amountsLength == minAmountOuts.length`, therefore the transaction does not revert. However, the `minAmount` that the user passed in for A is now applied to the redemption of D.

To be fair, this scenario is extremely unlikely, but I would still say that the issue is technically not fixed completely.

sogipec (Angle) commented:

> Thanks for raising this! We should have put a comment to show that we're aware of the fact that the issue is not technically and completely fixed. We're not going to add any additional code to fix it though, given the unlikeliness and the gas overhead it'd induce.

Disclosures

C4 is an open organization governed by participants in the community.

C4s audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.