Alongside
Security Review

Cantina Managed review by:
0xLeastwood, Lead Security Researcher
Zach Obront, Security Researcher

November 1, 2023

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

AMKT is a fully backed market index, providing exposure to a market-cap weighted basket of assets, to be reconstituted quarterly.

From Oct 23rd to Oct 27th the Cantina team conducted a review of amkt-v2 on commit hash b8822758. The team identified a total of **17** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 3
- Low Risk: 8
- Gas Optimizations: 1
- Informational: 4

3 Findings

3.1 High Risk

3.1.1 Governance delay in rebalancing will lead to exploited reconsitution process

Severity: High Risk

Context: CoreDeploy.s.sol#L90-L97, Config.sol#L16-L17

Description: The protocol is designed in such a way that, after deployment, governance will control rebalances:

- A bounty is proposed that includes all rebalance nominals, as well as the hash that will be added to `ActiveBounty.t.sol`.
- The vote will start after 1 day, remain open for 4 days, plus an additional 4 day delay before execution.
- 9 days after the rebalance nominals are made public, the rebalance will occur.

This creates a major risk for the protocol because, in the period between the rebalance being made public and the rebalance being executed, users are effectively given a free put and call option on the protocol's current constitution, with knowledge that it will adjust to the new constitution on a given date.

The Alongside team plans to address this issue by hedging against price movements for that 9 day period:

> The short/medium term fix is for fulfiller to “lock in” rebalancing trades as much as they can when they propose/submit the bounty assuming that their proposal will pass, to make sure they don’t suffer the full 9 days of price movement.

However, this solution is not sufficient, because the amount needed to hedge is variable. Because the trade required by the rebalancer is equal to `change in nominals * AMKT total supply`, any issuances and redemption over the course of the 9 days will change the amount of funds that should have been hedged.

Most importantly, market participants acting in self-interested ways will push the market in a direction that will siphon value from the rebalancing process.

The framework to think about this is that, as soon as the bounty is set, the future value of AMKT can implicitly be calculated from the new bounty amounts. But the market value of AMKT for issuance / redemptions continues to be variable at the old rates until the moment the rebalance happens.

This allows outside market participants to suck up all the value if the market moves in favor of the rebalanced portfolio, or AMKT holders to avoid all the downside if the market moves against the rebalanced portfolio, both of which have serious consequences for the protocol and rebalancer.

Proof of Concept:

There are two possible scenarios for us to explore. To simplify, let's say BTC is worth $30k and ETH is worth $1.5k, and the rebalance is moving from 0.01 BTC and 0 ETH nominals ($30k * 0.01 = $300 NAV per token) to 0 BTC and 0.2 ETH ($1.5k * 0.2 = $300 NAV per token). We can assume a total supply of 100 AMKT, for a total market cap of 100 * 300 = $30k.

On the day it's announced, this is a completely neutral trade. The rebalancer hedges by making a trade that represents trading 1 BTC for 20 ETH (worth $30k each).

Now, let's walk through two different scenarios:

BTC goes up vs. ETH (i.e. market moves in favor of current balance):

- BTC price goes to $40k, while ETH stays flat.
- The day before the rebalance, users have the ability to redeem() their AMKT for 0.01 BTC, which is worth $400.
- They know that after the trade, their shares will be worth 0.2 ETH = $300 (with the extra profits going to the rebalancer, from their perspective, but actually to the market maker who the hedge was with).
- All rational users will redeem their AMKT. Even if it's not all of them, clearly the incentive is to redeem and many of them will. Let's say this drops total supply down to 10 AMKT.

- Now there is only 0.1 BTC in the contract and it needs to be replaced with 2 ETH. But the rebalancer has already put out a large bet on the price movement.
- Regardless of the slight profit on the rebalance, they are out the $10k for their losses on the BTC vs ETH trade, whether they execute on the rebalance or not.

The result is that, if the market moves in favor of the current balance, the AMKT holders will redeem in advance of the rebalance for a free gain. This will crystalize the majority of the gains from the price movement, which is a problem if the rebalancer hedges, as they will lose funds on the hedge but not recoup them sufficiently on the rebalance transaction.

Thus, the rebalancer will have a small win if they don't hedge, and a large loss if they do hedge.

BTC goes down vs. ETH (i.e. market moves against current balance)

- BTC price goes down to $20k, while ETH stays flat.
- The day before the rebalance, users have the ability to issue() AMKT for 0.01 BTC, which is worth $200.
- They know that after the trade, their shares will be worth 0.2 ETH = $300 (with the losses accruing to the rebalancer, who we would think wouldn't do the trade, but since they've hedged, potentially will).
- Many outside arbitrageurs will issue AMKT. Let's say this increases the total supply to 1000 AMKT.
- Now there is 10 BTC in the contract that needs to be replaced with 200 ETH. This trade will cost $300k - $200k = $100k to execute, but the rebalancer has only earned $10k on their hedge.
- As a result, they will likely bail on the rebalancing. They will make a free $10k profit, but AMKT holders will have effectively given them a 9 day call option for free, and will be stuck with an inoptimal allocation at their expense.

The result is that, if the market moves in favor of the rebalanced portfolio, the market will issue a large amount of AMKT to take advantage of the free gain at the rebalancer's expanse, and the rebalancer will be stuck with such a large bill to perform the rebalance that they likely will not go through with it.

Thus, the rebalancer will end up bailing on the rebalance or suffering a large loss, whether they hedge or not.

Looking at these two scenarios, there is no rational behavior for the rebalancer that would allow anyone to perform that role without getting wrecked.

Recommendation: Any time gap between the bounty being made public and the rebalancing occurring opens up the opportunity for arbitrageurs to exploit the protocol. The only viable short term solution is to move rebalancing to be performed by the multisig, rather than governance.

Alongside: Fixed as recommended in PR 44 by using only the multisig controlled `ActiveBounty.sol` to set bounties.

Cantina Managed: Confirmed.

3.2 Medium Risk

3.2.1 Rounding in `tryInflation` will cause drift between vault balance and nominal calculation

Severity: Medium Risk

Context: Vault.sol#L149-L177

Description: When `tryInflation()` is called, the inflation amount is calculated as `seconds since last claim * inflation rate * AMKT supply`. This inflation value is used as the amount of AMKT to mint to the fee recipient.

In order to adjust the nominals to account for this increased total supply, each nominal is reduced in the following way:

```
uint256 valueMultiplier = fdiv(
    startingSupply,
    startingSupply + inflation
);

TokenInfo[] memory tokens = virtualUnits();
for (uint256 i = 0; i < tokens.length; i++) {
    _setNominal(
        SetNominalArgs({
            token: tokens[i].token,
            virtualUnits: fmul(tokens[i].units, valueMultiplier)
        })
    );
}
```

This lowering of the nominal value is subject to rounding, which could pose issues for tokens with fewer decimals and higher values. Given that the protocol's largest holding is WBTC (which has both a high value and only 8 decimals), this rounding will lead to drift between the calculated value based on nominals and the actual balance of the contract.

Note that this drift cannot be recouped. Any issuance or redemption are based on the nominal values, and all rebalancing pulls / pushes funds to the rebalancer to based on the nominal amounts. The extra balance in the contract will remain there indefinitely, and will not count towards AMKT market cap. It will effectively be burned.

Proof of Concept:

The following test demonstrates the issue, as well as estimating the total drift that will occur in WBTC over the course of a quarter:

```
function testInflationRoundingBTC() public {
    uint supply = 29370e18;

    uint btc_nominal = 214000;
    uint btc_starting_bal = btc_nominal * supply / 1e18;

    console2.log("Starting BTC Nominal: ", btc_nominal);
    console2.log("Starting BTC Balance: ", btc_starting_bal);

    uint inflation;
    uint new_supply;
    uint mult;
    for (uint i; i < 90; i++) {
        inflation = (supply * 304132280 * 1 days) / 1e18;
        new_supply = supply + inflation;
        mult = (supply * 1e18) / new_supply;
        btc_nominal = btc_nominal * mult / 1e18;
        supply = new_supply;
    }

    uint expected_btc_bal = btc_nominal * supply / 1e18;
    console2.log("New Expected Balance: ", expected_btc_bal);
    console2.log("Drift: ", btc_starting_bal - expected_btc_bal);
}
```

See the logs below:

```
Logs:
  Starting BTC Nominal:  214000
  Starting BTC Balance:  6285180000
  New Expected Balance:  6284164070
  Drift:  1015930
```

Each day when `tryInflation()` is called, the WBTC nominal will be rounded down slightly due to its low decimals. As a result, at the end of the quarter, the calculated balance of WBTC will be approximately 10^6 lower than the actual balance.

Based on the current BTC value of $35k (and assuming AMKT market cap doesn't change), we can estimate a drift `1e6/1e8 * 35000 = $350` per quarter that is locked in the contract.

Recommendation: The invariant check should adequately defend against the token's balance in the vault ever falling below the calculated value from the nominals. As a result, there are a number of possible fixes, for example:

1) In `fulfillBounty()`, rather than calling `transferFrom` to pull the calculated values from the rebalancer to the vault, allow the rebalancer to push those transfers, which will allow them to send a smaller amount in the case that there's excess balance.
2) Add a function for the multisig to rescue any tokens from the vault, as long as it leaves the contract in a state where `invariantCheck` passes.

Alongside: Acknowledged. We see the problem and plan to address it, but rather than rushing any code fix, we will plan to come up with a solution and will perform some surgery to rescue the tokens by upgrading the rebalancer or issuance contracts in the future.

Cantina Managed: Acknowledged.

3.2.2 `tryInflation` can be sandwiched to skirt fees (or profit, if there are AMKT lending markets)

Severity: Medium Risk

Context: Vault.sol#L149-L177

Description: When `tryInflation()` is called, all nominals are reduced slightly to account for the new tokens being minted.

However, if an AMKT holder redeems their AMKT tokens for underlying tokens in advance of the inflation occurring, and then issues new AMKT tokens after the inflation, they will be provided with a slight increase in their AMKT balance, equivalent to skirting the inflation that was applied to all other users.

While this may seem minimal (inflation is just 0.0025% per day), if a third party platform is set up that automates this for AMKT holders, it seems likely that many users would submit their tokens into this platform, dramatically decreasing the market cap of AMKT at each inflation call and therefore reducing the platform fees.

In an extreme case, if a lending market is to develop with a large amount of AMKT, it could lead to this same exploit being used to generate profit. In this case, the attacker would frontrun the inflation by borrowing the maximum amount of AMKT and redeeming it for all underlying tokens. After the inflation, they would issue the original amount of AMKT (which would leave a surplus of the underlying tokens for them), and return it to the lending platform.

The result would be free underlying tokens generated each day, which we can calculate as approximately 0.0025% per day. This amount would be quite small for a reasonable amount of tokens (and totally outweighed by price movements if it couldn't be sandwiched). But if 100,000 AMKT could be borrowed (value ~= $10mm), that would amount to $250 that could be earned each day on the arbitrage opportunity.

Recommendation: There is not a simple on chain solution for this problem that doesn't involve freezing issuance and redemption while inflation is occurring, which the team does not want to do.

The next best solution is to submit all inflation calls via Flashbots to ensure they can't be sandwiched.

Alongside: Acknowledged, `tryInflation()` will be called ONLY using Flashbots.

Cantina Managed: Acknowledged.

3.2.3 Bounty fulfillments may arbitrarily update the precision of the total basket of tokens

Severity: Medium Risk

Context: Issuance.sol#L37-L92

Description: AMKT consists of a basket of individual tokens and virtual units represent their allocation per `1e18` of each index token. When issuing tokens, the `underlyingAmount` is calculated as `fmul(tokens[i].units + 1, amount) + 1` where token units are stored in the same precision as the underlying token. As a result, it is possible to mint index tokens for essentially free by choosing specifically small amounts of index token to issue.

The profitability of such an action is limited to the degree at which several low precision tokens are replaced with tokens of high precision. In this example, USDC is replaced with ETH and what can be seen is that AMKT tokens were minted for near zero cost due to rounding down, and then after the new token's increased precision, they ultimately have slightly increased value.

```
contract BountyRoundingTest is StatefulTest {
    address fulfiller = address(bytes20(keccak256("fulfiller")));
    address user = address(bytes20(keccak256("user")));
    uint256 userMintAmount = 1e11;

    function testBug() public {
        // setup
        MockMintableToken USDC = new MockMintableToken("USDC", "USDC", 6, 0);
        MockMintableToken WETH = new MockMintableToken("WETH", "WETH", 18, 0);
        USDC.mint(fulfiller, 10000e6);
        WETH.mint(fulfiller, 10000e18);
        vm.startPrank(fulfiller);
        USDC.approve(address(bounty), 10000e6);
        WETH.approve(address(bounty), 10000e18);
        vm.stopPrank();
        USDC.mint(user, userMintAmount);
        // original bounty

        console.log(" ");
        console.log("initial state");
        console.log("Token supply: %s", indexToken.totalSupply());
        console.log(
            "User: USDC %s WETH %s",
            USDC.balanceOf(user),
            WETH.balanceOf(user)
        );
        console.log(
            "Fulfiller: USDC %s WETH %s",
            USDC.balanceOf(fulfiller),
            WETH.balanceOf(fulfiller)
        );
        console.log(
            "Vault: USDC %s WETH %s",
            USDC.balanceOf(address(vault)),
            WETH.balanceOf(address(vault))
        );

        TokenInfo[] memory originalTokens = new TokenInfo[](1);
        originalTokens[0] = TokenInfo(address(USDC), 1800e6);
        Bounty memory originalBounty = Bounty({
            infos: originalTokens,
            fulfiller: fulfiller,
            salt: keccak256("test"),
            deadline: block.timestamp + 100
        });
        bytes32 originalBountyHash = bounty.hashBounty(originalBounty);
        vm.prank(authority);
        activeBounty.setHash(originalBountyHash);
        vm.prank(fulfiller);
        bounty.fulfillBounty(originalBounty, false);

        console.log(" ");
        console.log("after original bounty fulfill");
        console.log(
            "User: USDC %s WETH %s",
            USDC.balanceOf(user),
            WETH.balanceOf(user)
        );
```

```
console.log(
    "Fulfiller: USDC %s WETH %s",
    USDC.balanceOf(fulfiller),
    WETH.balanceOf(fulfiller)
);
console.log(
    "Vault: USDC %s WETH %s",
    USDC.balanceOf(address(vault)),
    WETH.balanceOf(address(vault))
);

// mint
console.log(" ");
console.log("before user mint");
console.log(
    "User: USDC %s WETH %s",
    USDC.balanceOf(user),
    WETH.balanceOf(user)
);
console.log(
    "Fulfiller: USDC %s WETH %s",
    USDC.balanceOf(fulfiller),
    WETH.balanceOf(fulfiller)
);
console.log(
    "Vault: USDC %s WETH %s",
    USDC.balanceOf(address(vault)),
    WETH.balanceOf(address(vault))
);
vm.startPrank(user);
USDC.approve(address(issuance), userMintAmount);
issuance.issue(userMintAmount);
vm.stopPrank();
console.log(" ");
console.log("after user mint");
console.log(
    "User: USDC %s WETH %s",
    USDC.balanceOf(user),
    WETH.balanceOf(user)
);
console.log(
    "Fulfiller: USDC %s WETH %s",
    USDC.balanceOf(fulfiller),
    WETH.balanceOf(fulfiller)
);
console.log(
    "Vault: USDC %s WETH %s",
    USDC.balanceOf(address(vault)),
    WETH.balanceOf(address(vault))
);
// new bounty
TokenInfo[] memory newTokens = new TokenInfo[](2);
newTokens[0] = TokenInfo(address(USDC), 0);
newTokens[1] = TokenInfo(address(WETH), 1e18);
Bounty memory newBounty = Bounty({
    infos: newTokens,
    fulfiller: fulfiller,
    salt: keccak256("test"),
    deadline: block.timestamp + 100
});
bytes32 newBountyHash = bounty.hashBounty(newBounty);
vm.prank(authority);
activeBounty.setHash(newBountyHash);
vm.prank(fulfiller);
bounty.fulfillBounty(newBounty, false);

console.log(" ");
console.log("after new bounty fulfill");
console.log(
    "User: USDC %s WETH %s",
    USDC.balanceOf(user),
    WETH.balanceOf(user)
);
console.log(
    "Fulfiller: USDC %s WETH %s",
    USDC.balanceOf(fulfiller),
```

```
            WETH.balanceOf(fulfiller)
        );
        console.log(
            "Vault: USDC %s WETH %s",
            USDC.balanceOf(address(vault)),
            WETH.balanceOf(address(vault))
        );

        // redeem
        console.log(" ");
        console.log("before user redeem");
        console.log(
            "User: USDC %s WETH %s",
            USDC.balanceOf(user),
            WETH.balanceOf(user)
        );
        console.log(
            "Fulfiller: USDC %s WETH %s",
            USDC.balanceOf(fulfiller),
            WETH.balanceOf(fulfiller)
        );
        console.log(
            "Vault: USDC %s WETH %s",
            USDC.balanceOf(address(vault)),
            WETH.balanceOf(address(vault))
        );
        vm.prank(user);
        issuance.redeem(userMintAmount);
        console.log(" ");
        console.log("after user redeem");
        console.log(
            "User: USDC %s WETH %s",
            USDC.balanceOf(user),
            WETH.balanceOf(user)
        );
        console.log(
            "Fulfiller: USDC %s WETH %s",
            USDC.balanceOf(fulfiller),
            WETH.balanceOf(fulfiller)
        );
        console.log(
            "Vault: USDC %s WETH %s",
            USDC.balanceOf(address(vault)),
            WETH.balanceOf(address(vault))
        );
    }
}
```

Fortunately, this is not exploitable as the rounding is not significant enough to be profitable for a single individual, however, in total, the bounty fulfiller would have to cover the cost of any increased precision. The converse is true when total precision decreases for the index token, token holders are effectively losing out and the fulfiller profits slightly.

Recommendation: Ensure this is documented in the responsibility of the bounty fulfiller. The maximum impact on the bounty fulfiller can be predetermined as the new basket of tokens is known prior to a rebalance. It is their responsibility to ensure this is still profitable and satisfy the invariant check post bounty fulfilment. Some considerations also need to be made whether this rounding affects future issuance/redemptions.

Alongside: Acknowledged.

Cantina: Ultimately, this can be accounted for by the bounty fulfiller.

3.3 Low Risk

3.3.1 Ownership transfer risks in migration process

Severity: Low Risk

Context: CoreDeploy.s.sol#L123

Description: Over the course of the migration process, the vault's ownership changes hands a number of times:

- When `deployVault()` is called, owner is set to the `msg.sender` who deployed the contract.
- Later in CoreDeploy.s.sol, it is transferred to the multisig. However, this transfer is simply a proposal to transfer. The ownership is actually accepted by the multisig during the Gnosis Safe batch transaction.
- Later in the batch transaction, the Timelock is set as the pending owner. It will take ~9 days for governance to accept that ownership.

This creates two vulnerable points for centralization. During the time between vault deployment and migration, a single user has control over the vault. During the time between migration and the first governance proposal accepting that ownership, the multisig has complete control over the vault.

Recommendation: Users should be aware of the centralization of the protocol during this process.

Alongside: Acknowledged.

Cantina Managed: Acknowledged.

3.3.2 Inflation calculations start at deploy time, which could lead to temporary double fees

Severity: Low Risk

Context: Vault.sol#L84

Description: Inflation can be accrued any time more than `1 days` has passed by taking the number of seconds that have passed since `lastKnownTimestamp` and multiplying it by the inflation rate and the current supply of AMKT:

```
uint256 timestampDiff = block.timestamp - lastKnownTimestamp;
uint256 inflation = fmul(startingSupply, timestampDiff * inflationRate);
```

This `lastKnownTimestamp` is set to `block.timestamp` in the Vault's constructor function:

```
lastKnownTimestamp = block.timestamp;
```

Prior to the migration, inflation accrues on the AMKT token, which will stop when AMKT is migrated to the new version. Based on the deployment schedule, the Vault will be deployed a few days in advance of the AMKT migration. As a result, inflation may be double paid for that period.

Recommendation: Do not call `mintToFeeReceiver()` on the old AMKT contract between the time when the Vault is deployed and the time when AMKT is migrated.

Alongside: Acknowledged, this is a permissioned function and will not be called by our team during this period.

Cantina Managed: Acknowledged.

3.3.3 Tokens can be included multiple times in bounty, triggering `invariantCheck` or harming rebalancer

Severity: Low Risk

Context: Bounty.sol#L145-L153

Description: Tokens in a bounty are validated against the Vault's current underlying tokens to ensure that all existing tokens are explicitly included in the bounty.

```
address[] memory prevTokens = vault.underlying();

for (uint256 i; i < bounty.infos.length; i++) {
    if (i < prevTokens.length && prevTokens[i] != bounty.infos[i].token)
        revert BountyMustIncludeAllUnderlyings();
    if (bounty.infos[i].token == address(0))
        revert BountyInvalidToken();
}
```

This check confirms that the bounty begins by explicitly including each underlying token (in order), after which it can include new tokens.

When a token is included in the bounty, `(new nominal - old nominal) * AMKT total supply` is settled between the rebalancer and the vault, and then the nominal value is set to the new value. Note that the `old nominal` value is cached from before the function was called, and is not changed as nominals are updated.

The problem arises if a token is included in the bounty list twice. In this situation, the amount will be settled up between the rebalancer and the vault for each update individually (based on the previous nominal amount), but the new nominal value will just be set to the final value.

There are two possible consequences to this:

1) If the nominal of the token is decreasing, too many tokens will be sent from the vault to the rebalancer. As a result, the vault will fail the invariant check and the rebalancing will not be possible.
2) If the nominal of the token is increasing, too many tokens will be sent from the rebalancer to the vault. This will throw off the accounting of the vault, and leads to a buffer of tokens before the invariant check is triggered on future contract calls.

Recommendation: For maximum safety, add a check that each token in the bounty must be unique.

Since this expensive to perform on chain, it would also be acceptable to include in all governance proposals regarding the rebalancing an explicit mention that this check should be performed prior to approving a bounty.

Alongside: Acknowledged. This will be checked off chain before submitting a bounty.

Cantina Managed: Acknowledged.

3.3.4 Unbounded loops could cause DOS if `_underlying` gets too large

Severity: Low Risk

Context: Bounty.sol#L64-L109

Description: Many of the functions in the protocol include unbounded loops over the number of tokens in `_underlying`. While this number is currently set to just 15, in the event that the protocol is adapted to support a larger number of tokens, this could cause issues where the amount of gas needed for crucial protocol actions is greater than the block gas limit.

Proof of Concept:

The following test, when added to `Bounty.t.sol`, demonstrates that with 1650 tokens in the protocol, bounties can no longer be fulfilled because the loops cause the fulfillment cost to be greater than the block gas limit of 30 million:

```
function test__ReconstitutionOOG() public {
    TokenInfo[] memory tokens = seedInitial(1650);
    TokenInfo[] memory newTokens = new TokenInfo[](tokens.length);
    for (uint i = 0; i < tokens.length; i++) {
        newTokens[i] = TokenInfo({
            token: tokens[i].token,
            units: tokens[i].units + 100
        });
    }

    for (uint256 i = 0; i < newTokens.length; i++) {
        deal(newTokens[i].token, address(this), type(uint32).max);
        IERC20(address(newTokens[i].token)).approve(
            address(bounty),
            type(uint256).max
        );
    }

    Bounty memory _bounty = Bounty({
        infos: newTokens,
        fulfiller: address(0),
        salt: keccak256("test"),
        deadline: block.timestamp + 2 days
    });

    bytes32 _hash = bounty.hashBounty(_bounty);

    vm.prank(authority);
    activeBounty.setHash(_hash);

    uint before = gasleft();
    bounty.fulfillBounty(_bounty, true);
    console.log(before - gasleft());
}
```

See the log below:

```
Logs:
  30576417
```

Recommendation: Add an explicit check when adding elements to `_underlying` that its length is substantially less than 1650.

Alongside: Acknowledged, there is no intention to add such a large number of tokens.

Cantina Managed: Acknowledged.

3.3.5 Protocol governance initially has low token delegation

Severity: Low Risk

Context: IndexToken.sol

Description: On token upgrade, index token v2 must initially checkpoint the total supply to facilitate token redemptions by v1 token holders.

However, it is also necessary for these same holders to self delegate their token balance as this is typically done when minting tokens, but the v1 index token was not a governance token with voting functionality.

Recommendation: Ensure this is documented to users and necessary for a fully functioning governance system where proposal quorum can be realistically reached.

Alongside: Acknowledged.

Cantina Managed: The Alongside team will have sufficient tokens to meet quorum in the beginning and intend to encourage users to delegate their tokens as soon as the migration is complete.

3.3.6 Both `Bounty` deployments will be marked as `version 0`

Severity: Low Risk

Context: CoreDeploy.s.sol#L139-L154

Description: When the new contracts are deployed, we deploy two versions of `Bounty.sol`.

The first is attached to an `ActiveBounty.sol` contract with the `MULTISIG` as the authority. This is used for the initial bounty.

The second is attached to an `ActiveBounty.sol` contract with the `Timelock` as the authority. This will be used going forward after the first bounty is set.

There is a field in `Bounty.sol` that holds the version, which is used in bounty hashes. However, in both deployments, this value is set to the constant of `BOUNTY_VERSION`, which equals `0`.

```
function deployBounty(
    address _vault,
    address _bountySetter
) internal returns (InvokeableBounty, ActiveBounty) {
    ActiveBounty activeBounty = new ActiveBounty(_bountySetter);

    InvokeableBounty invokeableBounty = new InvokeableBounty({
        _vault: _vault,
        _activeBounty: address(activeBounty),
        _version: BOUNTY_VERSION,
        _chainId: 1
    });

    return (invokeableBounty, activeBounty);
}
```

Note that the version is not needed for the hash to be secure, so this doesn't pose a security risk. It simply reflects inaccurate information to have two bounties on chain actively being used, and being tagged as the same version.

Recommendation: Update the deploy script to deploy the first bounty as `version 0`, and the second as `version 1`.

Alongside: Fixed as recommended in PR 38.

Cantina Managed: In the end, another issue led to the removal of version from the bounty hash, which deemed this fix unnecessary.

3.3.7 `setInflationRate` will apply the new rate retroactively, potentially stealing funds

Severity: Low Risk

Context: Vault.so#L130-L136

Description: The `inflationRate` is a per second measurement used to calculate the amount of inflation to issue, as follows:

```
uint256 timestampDiff = block.timestamp - lastKnownTimestamp;
uint256 inflation = fmul(startingSupply, timestampDiff * inflationRate);
```

When `setInflationRate()` is called to adjust the inflation rate, the new rate is updated without any call to sync up the inflation before it happens. As a result, the new rate is applied retroactively to all the time since the `lastKnownTimestamp`:

```
function setInflationRate(uint256 _inflationRate) external only(owner()) {
    if (_inflationRate > SCALAR) {
        revert AMKTVaultInflationRateTooLarge();
    }
    inflationRate = _inflationRate;
    emit VaultInflationRateSet(_inflationRate);
}
```

Given that `tryInflation()` can only be called by the `feeRecipient` to sync up inflation, it is possible that this could represent a long lag, and apply a higher (or lower) inflation rate for a longer period of time than intended.

In the extreme case, the maximum inflation rate (`1e18 / second`) would result in 86400x the supply to be issued as inflation each day, which could easily dwarf the total supply and provide all the value of AMKT to the `feeRecipient`.

Recommendation: Add a call to `tryInflation()` at the beginning of `setInflationRate()`.

If there is a concern that this could cause other problems, a looser recommendation would be to skip the `tryInflation()` call when `emergency == true`, which would default to the correct behavior except in the event of an emergency.

Alongside: Acknowledged.

Cantina Managed: Acknowledged.

3.3.8 `IndexToken` requires larger gap to cover previously used slots

Severity: Low Risk

Context: IndexToken.sol

Description: The previous implementation of the AMKT token used storage slots up to 259 to store values, which we can observe with `cast storage 0x88f84864fd0839a7753199b01acb89c4714319f2 --rpc-url https://eth.llamarpc.com/`:

Name	Type	Slot	Offset	Bytes	Value
_initialized	uint8	0	0	1	1
_initializing	bool	0	1	1	1
__gap	uint256[50]	1	0	1600	0
_balances	mapping(address => uint256)	51	0	32	0
_allowances	mapping(address => mapping(address => uint256))	52	0	32	0
_totalSupply	uint256	53	0	32	0
_name	string	54	0	32	476862113492504397183254925688460240431103541505181257515933865131 37622056988
_symbol	string	55	0	32	476862113492504397183254925688460240431103541505181257515933865131 37622056988
__gap	uint256[45]	56	0	1440	0
_owner	address	101	0	20	833204653418720119172618286816296123974065860883
__gap	uint256[49]	102	0	1568	0
proposedOwner	address	151	0	20	0
__gap	uint256[50]	152	0	1600	0
_paused	bool	202	0	1	0
__gap	uint256[49]	203	0	1568	0
feeRatePerDayScaled	uint256	252	0	32	0
feeTimestamp	uint256	253	0	32	1665698027
feeReceiver	address	254	0	20	1
methodologist	address	255	0	20	0
minter	address	256	0	20	0
methodology	string	257	0	32	0
supplyCeiling	uint256	258	0	32	0
isRestricted	mapping(address => bool)	259	0	32	0

The updated storage layout (which can be accessed with `forge inspect IndexToken storage`) implements gaps and manual slot overrides to ensure the layouts line up, but only uses slots up to 253.

Name	Type	Slot	Offset	Bytes
_initialized	uint8	0	0	1
_initializing	bool	0	1	1
__gap	uint256[50]	1	0	1600
_balances	mapping(address => uint256)	51	0	32
_allowances	mapping(address => mapping(address => uint256))	52	0	32
_totalSupply	uint256	53	0	32
_name	string	54	0	32
_symbol	string	55	0	32
__gap	uint256[45]	56	0	1440
_HASHED_NAME	bytes32	101	0	32
_HASHED_VERSION	bytes32	102	0	32
__gap	uint256[50]	103	0	1600
_nonces	mapping(address => struct CountersUpgradeable.Counter)	153	0	32
_PERMIT_TYPEHASH_DEPRECATED_SLOT	bytes32	154	0	32
__gap	uint256[49]	155	0	1568
_delegates	mapping(address => address)	204	0	32
_checkpoints	mapping(address => struct ERC20VotesUpgradeable.Checkpoint[])	205	0	32
_totalSupplyCheckpoints	struct ERC20VotesUpgradeable.Checkpoint[]	206	0	32
__gap	uint256[47]	207	0	1504

You can see that there is a collision between `_HASHED_NAME` and `_HASHED_VERSION`. This does not cause any immediate damage, but in the event of a future upgrade, slots 254-259 will remain initialized. If these values are not overridden (which could easily be forgotten, given that the current contract does not use them), the result will be unexpected values.

Recommendation: Add a gap of 6 slots to the new implementation to protect against reusing slots in future upgrades.

Alongside: Fixed as recommended in PR 37.

Cantina Managed: Confirmed.

3.4 Gas Optimization

3.4.1 Bounty hash can be returned by `_validateInput` to reduce gas costs

Severity: Gas Optimization

Context: Bounty.sol#L68-L70

Description: In `fulfillBounty()`, we begin by calling `_validateInput()`, which hashes the bounty and validates it against `ActiveBounty.sol` to confirm that it is valid, the fulfiller is approved, and the deadline is not passed.

We then proceed to hash the bounty in `fulfillBounty()` in order to mark it as completed.

It is not necessary to perform this hash twice, and we can reuse it to save gas.

Recommendation: Return the bounty from `_validateInput()` directly, reusing it and saving the extra gas costs.

This will reduce gas costs for the rebalance from:

- Before: 353,403.

- After: 349,159.

Alongside: Fixed as recommended in PR 36.

Cantina Managed: Confirmed.

3.5 Informational

3.5.1 Missing natspec in `Vault` contract's constructor

Severity: Informational

Context: Vault.sol#L56-L67

Description: `Vault.sol`'s constructor function takes an `_emergencyResponder` address as an argument, which is assigned to the `emergencyResponder` storage variable.

However, the natspec for the function is missing documentation for this argument.

```
/// @notice Constructor
/// @param _indexToken The index token address
/// @param _owner The owner of the vault
/// @param _feeRecipient The recipient of the fee
/// @param _inflationRate The per second inflation rate
constructor(
    IIndexToken _indexToken,
    address _owner,
    address _feeRecipient,
    address _emergencyResponder,
    uint256 _inflationRate
) {
```

Recommendation: Add the relevant natspec to the function.

Alongside: Added in PR 39.

Cantina Managed: Confirmed.

3.5.2 AMKT is severely limited as a governance token, creating centralization risks for users

Severity: Informational

Context: IndexToken.sol

Description: Index v2 token has updated `ERC20` mechanics which allow token holders to vote on proposals which initially have ownership over the vault and token contracts.

AMKT is not a fully fledged governance token and only has the ability to indicate support on proposals, but ultimately the `multisig` has complete control over proposal vetoing and execution.

It is also possible for this `multisig` account to make proposals directly to the timelock controller contract and bypass any voting period.

This gives the multisig complete control over the protocol, with abilities such as:

- Setting bounties in such a way that the rebalancer steals value from AMKT token holders.
- Setting inflation rates in such a way that they inflate away all value from AMKT token holders.

Recommendation: Ensure there is a proper plan to transition to a protocol which is predominantly or completely controlled by token governance. Protocol actors are trusting the Alongside team to act honestly and not be compromised.

Alongside: Acknowledged.

Cantina Managed: Acknowledged.

3.5.3 `hashBounty` unnecessarily encodes redundant data

Severity: Informational

Context: Bounty.sol#L111-L125

Description: Bounty hashes are used to verify that the fulfiller provides the correct data as specified by the bounty authority.

`version` and `chainId` are immutable variables encoded alongside the bounty but do not provide any replay protection or useful information that is later used.

Recommendation: Consider removing `version` and `chainId` from the bounty hash computation.

Alongside: Fixed in PR 42.

Cantina Managed: Verified fix.

3.5.4 `_validateInput` and `_checkSupplyChange` can be marked as view functions

Severity: Informational

Context: Bounty.sol#L129-L153, Bounty.sol#L155-L159

Description: The `fulfillBounty()` function calls out to both `_validateInput()` and `_checkSupplyChange()` functions as it validates the bounty and rebalancing process. Both of these function perform not state changes, so can be marked as `view`.

Recommendation: Add the `view` visibility to both functions.

Alongside: FIxed as recommended in PR 36.

Cantina Managed: Confirmed.