Cryptex
Security Review

Cantina Managed review by:
Patrickd, Security Researcher
Cccz, Security Researcher

September 27, 2024

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).

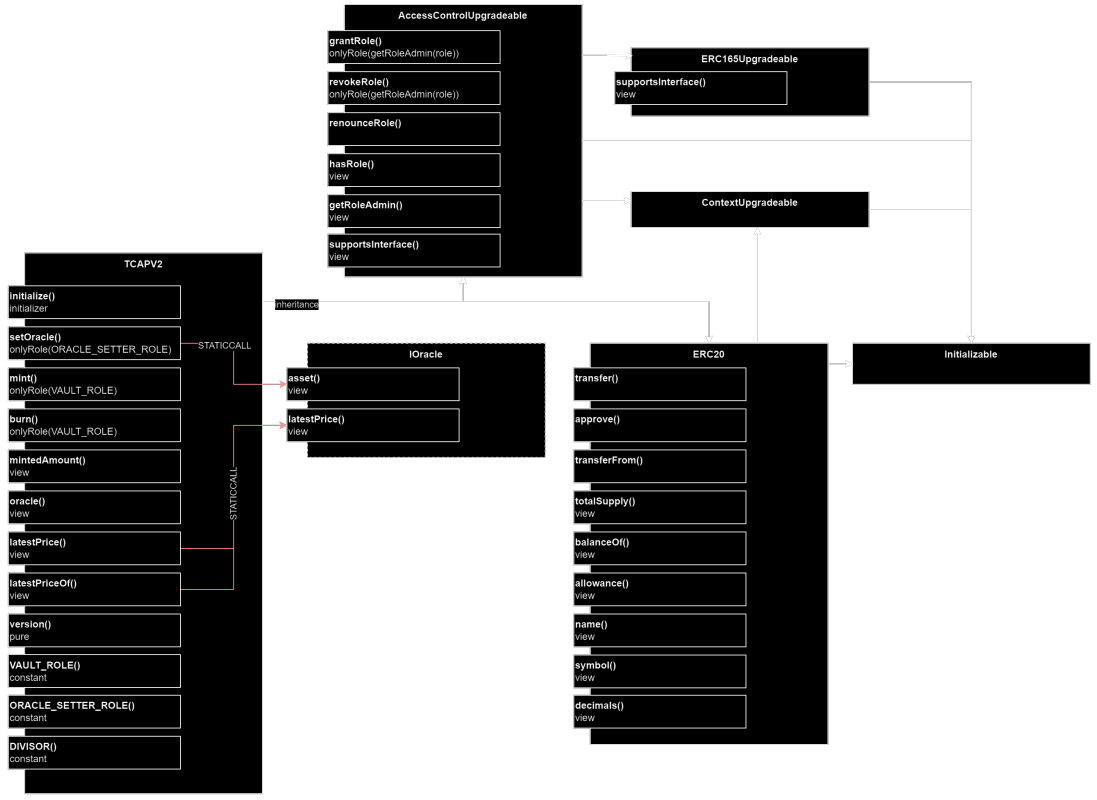
2 Security Review Summary

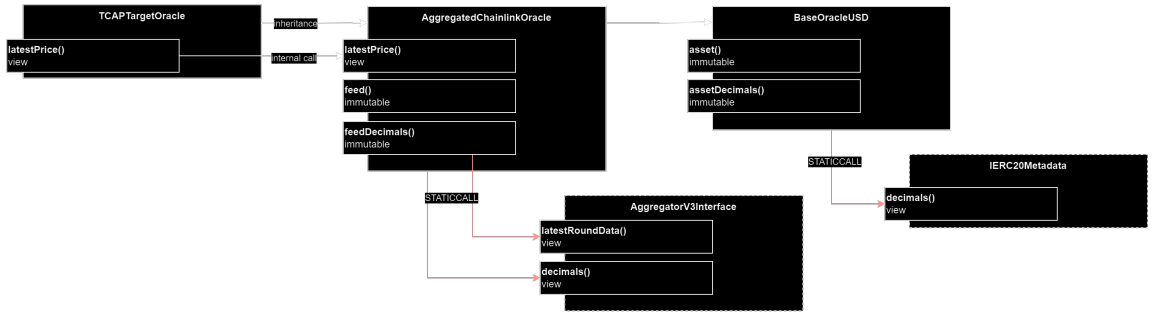
Cryptex brings together a comprehensive suite of DeFi services into one intuitive platform, offering a seamless and powerful trading experience.

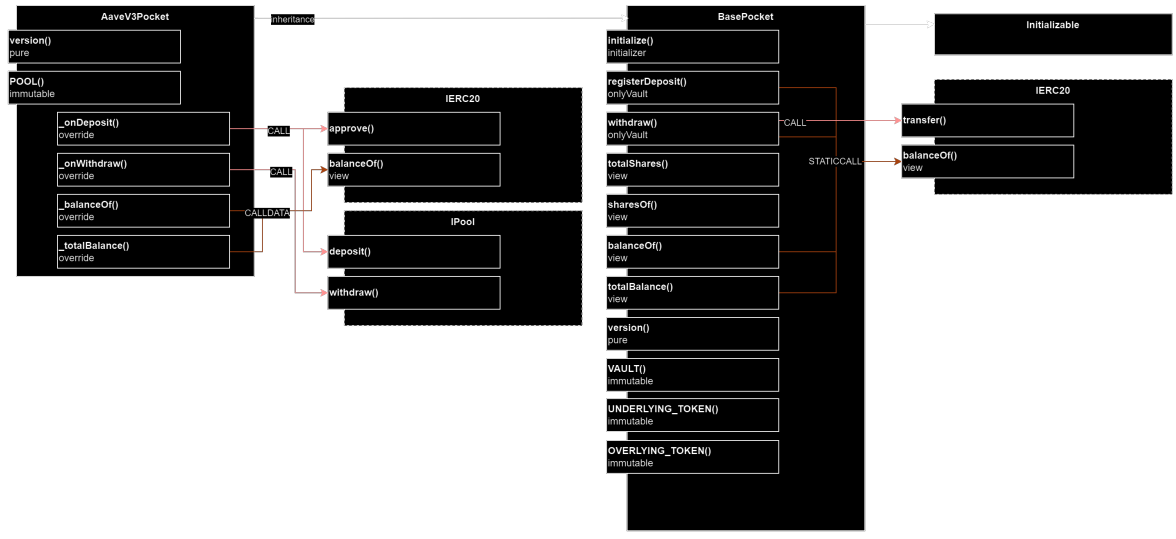
From Sep 16th to Sep 24th the Cantina team conducted a review of cryptex-monorepo on commit hash 459e49fd. The team identified a total of **18** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 4
- Medium Risk: 1
- Low Risk: 5
- Gas Optimizations: 2
- Informational: 6

3 Cryptex architecture








AccessControlUpgradeable
grantRole()
onlyRole(getRoleAdmin(role))
revokeRole()
onlyRole(getRoleAdmin(role))
renounceRole()
hasRole()
view
getRoleAdmin()
view
supportsInterface()
view
ERC165Upgradeable
supportsInterface()
view
Initializable
ContextUpgradeable
Vault
initialize()
initializer
addPocket()
onlyRole(POCKET_SETTER_ROLE)
disablePocket()
onlyRole(POCKET_SETTER_ROLE)
updateInterestRate()
onlyRole(FEE_SETTER_ROLE)
updateFeeRecipient()
onlyRole(FEE_SETTER_ROLE)
updateOracle()
onlyRole(ORACLE_SETTER_ROLE)
updateLiquidationParams()
onlyRole(LIQUIDATION_SETTER_ROLE)
deposit()
depositWithPermit()
withdraw()
mint()
burn()
liquidate()
healthFactor()
view
collateralValueOf()
view
collateralValueOfUser()
view
mintedValueOf()
view
mintedValueOfUser()
view
collateralOf()
view
mintedAmountOf()
view
outstandingInterestOf()
view
latestPrice()
view
oracle()
view
interestRate()
view
feeRecipient()
view
liquidationParams()
view
pockets()
view
pocketEnabled()
view
version()
pure
TCAPV2()
immutable
COLLATERAL()
immutable
POCKET_SETTER_ROLE()
constant
FEE_SETTER_ROLE()
constant
ORACLE_SETTER_ROLE()
constant
LIQUIDATION_SETTER_ROLE()
constant
Internal calls
STATICCALL
CALL
inheritance
Multicall
multicall()
payable
DELEGATEACALL
IPocket
VAULT()
view
UNDERLYING_TOKEN()
view
registerDeposit()
IOracle
asset()
view
SafeTransferLib for address(IERC20)
transferFrom()
via safeTransferFrom()
IPermit2
permitTransferFrom()
withdraw()
ITCAPV2
mint()
burn()
latestPrice()
view
assetDecimals()
view
balanceOf()
view

4 Findings

4.1 High Risk

4.1.1 Calculating liquidationReward without considering collateral decimals

Severity: High Risk

Context: LiquidationLib.sol#L17-L19

Description: In liquidation, the `liquidationReward()` function is used to calculate the liquidator's reward, and it will calculate the rewarded collateral amount based on the TCAP amount burned by the liquidator.

```
function liquidationReward(uint256 burnAmount, uint256 tcapPrice, uint256 collateralPrice, uint64
↪  liquidationPenalty) internal pure returns (uint256) {
    return burnAmount * tcapPrice * (1e18 + liquidationPenalty) / collateralPrice / 1e18;
}
```

The problem here is that `liquidationReward()` doesn't consider the decimals of the collateral and TCAP, which results in the calculated liquidationReward being quite large when the collateral is a low decimals token such as USDC/USDT, thus giving the liquidator a larger reward.

Recommendation: Change to

```
- liquidationReward = LiquidationLib.liquidationReward(burnAmount, tcapPrice, collateralPrice,
↪  liquidation.penalty);
+ liquidationReward = LiquidationLib.liquidationReward(burnAmount, tcapPrice, collateralPrice,
↪  liquidation.penalty, assetDecimals);
  // ...
- function liquidationReward(uint256 burnAmount, uint256 tcapPrice, uint256 collateralPrice, uint64
↪  liquidationPenalty) internal pure returns (uint256) {
-     return burnAmount * tcapPrice * (1e18 + liquidationPenalty) / collateralPrice / 1e18;
+ function liquidationReward(uint256 burnAmount, uint256 tcapPrice, uint256 collateralPrice, uint64
↪  liquidationPenalty, uint8 collateralDecimals) internal pure returns (uint256) {
+     return burnAmount * tcapPrice * (1e18 + liquidationPenalty) * 10 ** collateralDecimals / collateralPrice
↪  / 1e18 / 10 ** 18;
  }
```

Cryptex: Fixed in PR 9.

Cantina Managed: Fixed.

4.1.2 Depositing to Pocket is vulnerable to inflation attacks

Severity: High Risk

Context: BasePocket.sol#L50

Description: When depositing to Pocket, a malicious user can manipulate pricePerShare by donating, so that other users will lose due to rounding down when depositing. Consider the following scenario:

1. Alice deposits 1e18 wei collateral to Pocket.
2. Bob observes Alice's transaction, and frontruns Alice with the following action.
3. Bob deposits 1 wei collateral and mint 1 wei share.
4. Bob transfers 1e18 wei collateral to Pocket, now total assets are 1e18 + 1 wei, and total shares are 1 wei.
5. Alice's transaction is executed, `shares = 1 * 1e18 / (1e18 + 1)`, rounding down to 0, Alice receives 0 share.
6. Bob withdraws 1 wei share and receives 2e18+1 wei collateral.

Recommendation: According to OZ's explanation of ERC-4626 inflation attacks, there are several recommendations to mitigate inflation attack, such as:

1. The use of an ERC4626 Router does not resolve the issue on its own, as it relies on users to perform slippage control during mint/deposit to prevent losses.

```
function deposit(
    IERC4626 vault,
    address to,
    uint256 amount,
    uint256 minSharesOut
) public payable virtual override returns (uint256 sharesOut) {
    if ((sharesOut = vault.deposit(amount, to)) < minSharesOut) { // @audit: slippage control here
        revert MinSharesError();
    }
}
```

2. Tracking assets internally instead of relying on current token balances. In other words, this requires the protocol to not use `balanceOf()` to track assets, which does not apply to rebase tokens, such as aToken.
3. Dead Shares (like Uniswap V2).

```
function mint(address to) external lock returns (uint liquidity) {
    // ...
    uint _totalSupply = totalSupply; // gas savings, must be defined here since totalSupply can update
↪  in _mintFee
    if (_totalSupply == 0) {
        liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
       _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tokens
```

   After we mint the initial 1000 shares to `address(0)`, when the attacker makes a donation, the donated assets will increase the value of these 1000 dead shares, causing the attacker to lose. Consider that the attacker mints 1001 shares, 1000 shares are minted to `address(0)`, and 1 share is minted to the attacker. After that, the attacker donates `1e18` collateral. The attacker immediately loses `1000/1001 * 1e18` collateral.

4. Virtual Shares and Decimals Offset.

   This is the method used by OZ ERC4626 and is proven mathematically within OpenZeppelin's documentation. However, this method may significantly increase `totalShares`, so the `assert()` statement in `BasePocket.registerDeposit()` may need to be changed if it is used.

```
assert($.sharesOf[user] < 1e38);
```

Cryptex: Fixed in commit 18949206.

Cantina Managed: Fixed.

4.1.3 Calculating interest without considering the collateral decimals

Severity: High Risk

Context: Vault.sol#L289-L293

Description: The `outstandingInterestOf()` function converts the interest from the TCAP amount to the collateral amount. However, it does not consider TCAP and collateral decimals when converting, which results in a rather large interest calculated when using low-decimal tokens such as USDC/USDT as collateral.

```
function outstandingInterestOf(address user, uint96 pocketId) public view returns (uint256) {
    MintData storage $ = _getVaultStorage().mintData;
    uint256 interestAmount = $.interestOf(_toMintId(user, pocketId));
    return interestAmount * TCAPV2.latestPrice() / latestPrice();
}
```

Proof of concept: The proof of concept shows that when collateral is 6 decimals, `outstandingInterest()` returns `0.1e18` (should be `0.1e6`), and `collateralOf` will underflow:

```
function test_poc() public {
    address user = makeAddr("user");
    uint256 amount = 100e6;
    deposit(user, amount);
    vm.prank(user);
    vault.mint(pocketId, 10e18);
    uint timestamp = 365 days;
    vm.warp(timestamp);
    uint256 outstandingInterest = vault.outstandingInterestOf(user, pocketId);
    console.logUint(outstandingInterest); // 0.1e18
    console.logUint(vault.collateralOf(user, pocketId)); // [FAIL. Reason: panic: arithmetic underflow or
↪  overflow (0x11)]
}
```

Recommendation: Considering collateral and TCAP decimals in `outstandingInterestOf()`

```
  function outstandingInterestOf(address user, uint96 pocketId) public view returns (uint256) {
      MintData storage $ = _getVaultStorage().mintData;
      uint256 interestAmount = $.interestOf(_toMintId(user, pocketId));
+     uint256 assetDecimals = _getVaultStorage().oracle.assetDecimals();
+     return interestAmount * TCAPV2.latestPrice() * 10 ** assetDecimals / latestPrice() / 10 ** 18;
-     return interestAmount * TCAPV2.latestPrice() / latestPrice();
  }
```

Cryptex: Fixed in PR 9.

Cantina Managed: Fixed.

4.1.4 Liquidated person's debts are not reduced in liquidation

Severity: High Risk

Context: Vault.sol#L245-L246

Description: After liquidation, the liquidator's TCAPV2 is burned to repay the liquidated person's debts. However, during liquidation, the `modifyPosition()` function is not called to update the position of the liquidated person to reduce the debt, which would result in the liquidated person being able to be liquidated repeatedly until the collateral goes to 0:

```
pocket.withdraw(user, liquidationReward, msg.sender);
TCAPV2.burn(msg.sender, burnAmount);
emit Liquidated(msg.sender, user, pocketId, liquidationReward, burnAmount);
```

Proof of concept: The proof of concept shows that after liquidation, the collateral is 0 and the debt remains unchanged:

```
function test_POC() public {
    address user = address(0xb0b);
    uint amount = 1e18;
    vm.assume(user != address(0) && user != address(vaultProxyAdmin));
    uint256 depositAmount = deposit(user, amount);
    vm.assume(depositAmount > 0);
    vm.prank(user);
    uint256 mintAmount = bound(amount, 1, depositAmount);
    vault.mint(pocketId, mintAmount);
    uint256 collateralValue = vault.collateralValueOfUser(user, pocketId);
    console.logUint(collateralValue); // 1000e18
    uint256 mintValue = vault.mintedValueOf(mintAmount);
    console.logUint(mintValue); // 1000e18

    uint256 multiplier = mintValue * 10_000 / (collateralValue);
    vm.assume(multiplier > 1);
    feed.setMultiplier(multiplier - 1);
    tCAPV2.mint(address(this), mintAmount);
    vm.expectEmit(true, true, true, true);
    emit IVault.Liquidated(address(this), user, pocketId, depositAmount, mintAmount);
    console.logUint(vault.mintedValueOfUser(user,pocketId)); // 1000e18
    collateralValue = vault.collateralValueOfUser(user, pocketId);
    console.logUint(collateralValue); // 99.9e18
    vault.liquidate(user, pocketId, mintAmount);
    collateralValue = vault.collateralValueOfUser(user, pocketId);
    console.logUint(collateralValue); // 0
    console.logUint(vault.mintedValueOfUser(user,pocketId)); // 1000e18
}
```

Recommendation: Change to

```
+ $.modifyPosition(_toMintId(user, pocketId), -burnAmount.toInt256());
  pocket.withdraw(user, liquidationReward, msg.sender);
  TCAPV2.burn(msg.sender, burnAmount);
  emit Liquidated(msg.sender, user, pocketId, liquidationReward, burnAmount);
```

Cryptex: Fixed in commits 50c7925a and 441a8137.

Cantina Managed: Fixed.

4.2 Medium Risk

4.2.1 Incorrect calculation of interests when taking fee

Severity: Medium Risk

Context: Vault.sol#L331-L333

Description: In _takeFee(), if the interest is greater than the collateral, the collateral is taken as interest.

```
function _takeFee(IPocket pocket, address user, uint96 pocketId) internal {
    uint256 interest = outstandingInterestOf(user, pocketId);
    uint256 collateral = collateralOf(user, pocketId);
    if (interest > collateral) interest = collateral;
```

The problem here is that the collateralOf() function already considers the interest, if the interest is greater than the balance, collateralOf() will underflow. And even if the balance is enough to cover the interest, the interest will be lower.

```
function collateralOf(address user, uint96 pocketId) public view returns (uint256) {
    IPocket pocket = _getVaultStorage().pockets[pocketId].pocket;
    return pocket.balanceOf(user) - outstandingInterestOf(user, pocketId);
}
```

For example, if the user has 15 USD balance and 10 USD interest, in `_takeFee()`, `collateralOf()` is `15 - 10 = 5` USD, which means that only 5 USD interest will be charged, even if the 15 USD balance is enough to pay 10 USD interest.

Recommendation: Change to

```
  function _takeFee(IPocket pocket, address user, uint96 pocketId) internal {
      uint256 interest = outstandingInterestOf(user, pocketId);
+     uint256 collateral = pocket.balanceOf(user);
-     uint256 collateral = collateralOf(user, pocketId);
      if (interest > collateral) interest = collateral;
      VaultStorage storage $ = _getVaultStorage();
      address feeRecipient_ = $.feeRecipient;
      if (interest != 0 && feeRecipient_ != address(0)) {
          pocket.withdraw(user, interest, feeRecipient_);
      }
      $.mintData.resetInterestOf(_toMintId(user, pocketId));
```

Also, since `_takeFee()` will only be called in `withdraw()`/`liquidate()`, which prevents the fee from being collected in time, it is recommended to wrap `_takeFee()` with a public function so that anyone can call it to collect the fee.

Cryptex: Fixed in commit d1799f6e.

Cantina Managed: Fixed.

4.3 Low Risk

4.3.1 Accumulated fee may be lost

Severity: Low Risk

Context: Vault.sol#L336-L339

Description: In _takeFee(), when interest is 0, interset is reset directly.

```
function _takeFee(IPocket pocket, address user, uint96 pocketId) internal {
    uint256 interest = outstandingInterestOf(user, pocketId);
    uint256 collateral = collateralOf(user, pocketId);
    if (interest > collateral) interest = collateral;
    VaultStorage storage $ = _getVaultStorage();
    address feeRecipient_ = $.feeRecipient;
    if (interest != 0 && feeRecipient_ != address(0)) {
        pocket.withdraw(user, interest, feeRecipient_);
    }
    $.mintData.resetInterestOf(_toMintId(user, pocketId));
}
```

Considering the fee is 5%, according to the calculation, the accumulated interest per second is `mintAmount * feeData.fee /(365 days * Constants.MAX_FEE) = mintAmount * 0.05 / 365 days = mintAmount / 630720000`, that is, when `mintAmount < 630720000 wei`, the accumulated interest per second will round down to 0. If the user calls `_takeFee()` once per second, the interest paid will always be 0.

```
function interestOf(Vault.MintData storage $, uint256 mintId) internal view returns (uint256 interest) {
    return $.deposits[mintId].accruedInterest + outstandingInterest($, feeIndex($), mintId);
}
// ...
function feeIndex(Vault.MintData storage $) internal view returns (uint256) {
    return $.feeData.index + (block.timestamp - $.feeData.lastUpdated) * $.feeData.fee * MULTIPLIER() / (365
↪  days * Constants.MAX_FEE);
}
// ...
function outstandingInterest(Vault.MintData storage $, uint256 index, uint256 mintId) private view returns
↪  (uint256 interest) {
    uint256 userIndex = $.deposits[mintId].feeIndex;
    return $.deposits[mintId].mintAmount * (index - userIndex) / MULTIPLIER();
}
```

In addition, the interest will be reset when `feeRecipient_` is `address(0)`.

Recommendation: Change to

```
  if (interest != 0 && feeRecipient_ != address(0)) {
      pocket.withdraw(user, interest, feeRecipient_);
+     $.mintData.resetInterestOf(_toMintId(user, pocketId));
  }
- $.mintData.resetInterestOf(_toMintId(user, pocketId));
```

Cryptex: Fixed in commit d1799f6e.

Cantina Managed: Fixed.

4.3.2 Lack of integration with Aave's reward distribution

Severity: Low Risk

Context: AaveV3Pocket.sol#L19-L30

Description: Aave v3 has an integrated Rewards Distribution System to incentivise the supply of liquidity, where deposited underlying tokens may yield rewards that can be obtained via the `RewardController` contract.

The AaveV3Pocket currently has no way to access any accumulated rewards to distribute them fairly among its users.

Recommendation: Consider integrating the `AaveV3Pocket` contract with Aave's `RewardController`.

Cryptex: Acknowledged. This is acceptable as-is for the moment. Should the need arise, rewards could be theoretically retrieved and distributed at a later point through an upgrade of the pocket contract, should those rewards turn out to be meaningful.

Cantina Managed: Acknowledged.

4.3.3 Harden Chainlink Price Feed integration

Severity: Low Risk

Context: AggregatedChainlinkOracle.sol#L18-L21

Description: The AggregatedChainlinkOracle has no validation of the returned data at the moment, aside from a simple sanity check on the price being non-zero.

```
function latestPrice() public view virtual override returns (uint256) {
    (, int256 answer,,,) = feed.latestRoundData();
    // @audit in case of a stale oracle do not revert because it would prevent users from withdrawing
    assert(answer > 0);
    // @audit feed decimals cannot exceed 18
    uint256 adjustedDecimalsAnswer = uint256(answer) * 10 ** (18 - feedDecimals);
    return adjustedDecimalsAnswer;
}
```

The Cryptex Team wishes for users to be able to withdraw any free funds at any time, and it also prefers having functions that improve protocol health, such as the burning of debt and liquidations, to be available at all times. Adding checks for the freshness of price information may impede these intentions by causing these actions to revert.

Officially Chainlink "encourages" consumers to check whether received data is fresh by inspecting `updatedAt` (or alternatively `answeredInRound`, but that has been depricated as answers no longer take multiple rounds to be computed). In practice, many protocols omit such checks completely, as done here. Others implement a wide variety of staleness threshold checks on updatedAt: Some use constant thresholds with values such as 5m, 90m, 4h, and even 24h. Some have the threshold configurable via storage.

There is no obvious consensus on whether and how to check for feed price "freshness". There's also a variety of sanity checks that projects implement similar to the `assert(answer > 0);` done here:

- `assert(roundId > 0)`.
- `assert(updatedAt > 0)`.
- `assert(updatedAt <= block.timestamp)`.

That it's indeed safe to ignore `startedAt` and `answeredInRound` can be verified by looking at Chainlink's contracts that simply return the values for `updatedAt` and `roundId` respectively in their place.

While there's arguably little harm to implementing the mentioned sanity checks, threshold checks on `updatedAt` or the price need to be decided based on the project's nature and risks.

Risk Analysis: As TCAP is a synthetic asset, there's arguably little immediate risk with its oracle price being outdated - it remains at the target price that its incentives attempt to peg to. Collateral prices being stale, on the other hand, could lead to "arbitrage" opportunities: Say the collateral is ETH and the price reported by the oracle remains stale at a high price while the market price of ETH has actually significantly decreased. Buying ETH from an AMM, depositing it into a Vault, minting TCAP, immediately selling it for ETH again, repeat - things like this may become profitable. The older Cryptex proposal CIP-14 may serve as an example precedent for such opportunities being exploited.

Aside from possible risks from prices being stale, there are risks associated with prices being updated after having been stale for a while. While a stale oracle target price for TCAP might be harmless, a sudden increase in TCAP price once the total market cap oracle is being updated again might surprise many borrowers with sudden liquidations. On the flip side, borrowers following market price movements attentively may use the time before the price feeds are updated as an opportunity to jump ship before reality hits chain, while less sophisticated borrowers will suffer.

On L2s such situations may end up being especially problematic: Many L2s, like this protocol's target chain Arbitrum, allow the submission of transaction even while the sequencer is down. It's not far fetched to assume that liquidators tend to be more technically sophisticated than the average borrower. It's likely that liquidators will be able to exploit this fact to liquidate borrowers who weren't able to access the protocol during the sequencer downtime to ensure sufficient collateral. Some protocols such as Aave have explicitly implemented "borrower grace periods" during which neither liquidations nor further borrowing is possible. These grace periods are automatically triggered by the contracts checking Chainlink's L2 Sequencer Uptime Feeds and intend to give borrowers some time to secure their positions once the sequencer is back online.

Recommendations:

- Add the above mentioned sanity checks.
- Implement monitoring for Chainlink pricefeeds and Arbitrum sequencer to be aware of any irregularities.
- Implement monitoring to detect any "arbitrage opportunities" and alert the team of any strong deviations between oracle and market prices.
- Consider implementing grace periods for borrowers after a sequencer outage.
- Encourage borrowers to subscribe to communication channels that will notify them if positions may be in danger due to oracle or sequencer outages.
- Document and publish possible ways to respond to such incidents for the governance community to consult.
- Consider implementing circuit breakers at least for extreme cases of price staleness or deviations. In the rare events that these trigger, or are about to trigger, a governance proposal may deploy adjusted oracle contracts with adjusted parameters depending on the situation.
- Alternatively to circuit breakers that pause the protocol entirely, consider at least implementing "speed bumps" which, when triggered during time of price staleness, price de-peg, or volatility, will prohibit the use of flash loans. The above mentioned "arbitrage opportunities" often rely on the use of flash loans to immediately deposit and mint within a single transaction. Active speed bump logic would prevent the use of flash loans by forcing the block number between the call of the deposit and the call of the mint function to differ.
- In case price checks will be restricted to certain interactions with the protocol, it's recommended to at least add them to the mint function, preventing the creation of new debt during unsafe periods. Not disabling liquidations may be in favor of the protocol, but it can cause borrowers to be unfairly liquidated, especially during times where only either of the oracles (collateral price or TCAP target price) becomes stale. Allowing the withdrawal of "free" collateral while price oracles are stale may at worst allow the protocol to temporarily have bad debt once oracles are back up. With the Cryptex Team's desire to allow withdrawals at any time in favor of the users, the governance should consider preparing emergency funds to buy such bad debt before it can negatively impact the TCAP price peg.

Cryptex: Fixed in commit 3209c762.

Cantina Managed: Fixed. The Cryptex Team addressed this finding by adding a staleness check that is only executed when attempting to mint TCAP (i.e. when creating new debt).

4.3.4 Correct overlying aToken can be determined automatically

Severity: Low Risk

Context: AaveV3Pocket.sol#L14-L16

Description: When initializing an AaveV3Pocket contract, the contract receives

- `underlyingToken_`: The address of the underlying ERC20 token, e.g. DAI.
- `overlyingToken_`: The address of the overlying ERC20 aToken belonging to Aave, e.g. aDAI.
- `aavePool`: The address of the Aave Pool to interact with.

Situations like these allow for mistakes to happen where the underlying and overlying tokens are actually incompatible.

Recommendation: It's possible to automatically determine the correct aToken address during the initialization of `AaveV3Pocket`:

```
- constructor(address vault_, address underlyingToken_, address overlyingToken_, address aavePool)
↪  BasePocket(vault_, underlyingToken_, overlyingToken_) {
+ constructor(address vault_, address underlyingToken_, address aavePool)
+     BasePocket(vault_, underlyingToken_, IPool(aavePool).getReserveData(underlyingToken_).aTokenAddress) {
      POOL = IPool(aavePool);
+     require(address(UNDERLYING_TOKEN) != address(0));
  }
```

Cryptex: Fixed in PR 13.

Cantina Managed: The Cryptex Team has addressed this issue within the attached pull request.

4.3.5 Use of deprecated Aave V3 Pool deposit function

Severity: Low Risk

Context: AaveV3Pocket.sol#L19-L21

Description: The AaveV3Pocket contract integrates with Aave V3 in order to allow debtors, willing to take on more risk, to make use of their collateral more efficiently.

To interact with this party, the protocol makes use of the `IPool` interface with contains the following documentation regarding the use of its `deposit()` function as done within `AaveV3Pocket`:

```
/**
 * @notice Supplies an `amount` of underlying asset into the reserve, receiving in return overlying aTokens.
 * - E.g. User supplies 100 USDC and gets in return 100 aUSDC
 * @dev Deprecated: Use the `supply` function instead
 * [...]
 */
function deposit(address asset, uint256 amount, address onBehalfOf, uint16 referralCode) external;
```

Recommendation: Replace the call to the deprecated `deposit()` function with a call to the equivalent `supply()` function instead:

```
- POOL.deposit(address(UNDERLYING_TOKEN), amountUnderlying, address(this), 0);
+ POOL.supply(address(UNDERLYING_TOKEN), amountUnderlying, address(this), 0);
```

Cryptex: Fixed in PR 13.

Cantina Managed: The Cryptex Team has addressed this issue within the attached pull request.

4.4 Gas Optimization

4.4.1 Public constants needlessly pollute function selection

Severity: Gas Optimization

Context: DeployTCAP.s.sol#L36, TCAPTargetOracle.sol#L13, TCAPV2.sol#L21-L24, Vault.sol#L62-L65

Description: At the moment the Vault and TCAPV2 contracts expose public getter functions for constant variable values.

```
bytes32 public constant POCKET_SETTER_ROLE = keccak256("POCKET_SETTER_ROLE");
bytes32 public constant FEE_SETTER_ROLE = keccak256("FEE_SETTER_ROLE");
bytes32 public constant ORACLE_SETTER_ROLE = keccak256("ORACLE_SETTER_ROLE");
bytes32 public constant LIQUIDATION_SETTER_ROLE = keccak256("LIQUIDATION_SETTER_ROLE");
```

```
bytes32 public constant VAULT_ROLE = keccak256("VAULT_ROLE");
bytes32 public constant ORACLE_SETTER_ROLE = keccak256("ORACLE_SETTER_ROLE");
uint256 public constant DIVISOR = 1e10;
```

These values being constant, and therefore accessible within code without the need of a getter function, arguably pollutes the function selector logic with unnecessary branches.

Recommendation: Consider making the function selection more efficient by making these constant variables internal such that they will no longer have a public getter function exposed.

Cryptex: Fixed in commit 892f192a.

Cantina Managed: Fixed.

4.4.2 Various small gas optimizations

Severity: Gas Optimization

Context: AaveV3Pocket.sol#L20, BasePocket.sol#L65, Multicall.sol#L15-L19, TCAPV2.sol#L57-L63, Vault.sol#L149, Vault.sol#L200

Description: The biggest gas inefficiencies are arguably due to the way the code is currently organized, especially around the fee/interest rate logic, causing a lot of gas being spent on redundant storage reads and writes. The current structure provides great testability, but makes significant gas optimizations rather difficult.

Although gas may not be the greatest concern with this project being primarily developed for layer 2 chains, the following are recommendations of changes that require few code adjustments but offer some efficiency benefits.

Recommendations:

- Vault.sol: The `liquidation()` function currently computes the `healthFactor_` as part of its business logic. This same computation is repeated when it goes on to call the `tokensRequiredForTargetHealthFactor()` function twice.

```
  /// @inheritdoc IVault
  function liquidate(address user, uint96 pocketId, uint256 burnAmount) external returns (uint256
↪  liquidationReward) {
  // ...
      uint256 healthFactor_ = LiquidationLib.healthFactor(mintAmount, tcapPrice, collateralAmount,
↪  collateralPrice, assetDecimals);
      if (healthFactor_ >= liquidation.threshold) revert LoanHealthy();
  // ...
      } else {
          uint256 minBurnAmount = LiquidationLib.tokensRequiredForTargetHealthFactor(
              liquidation.threshold + liquidation.minHealthFactor,
              mintAmount,
              tcapPrice,
              collateralAmount,
              collateralPrice,
              liquidation.penalty,
              assetDecimals
          );
  // ...
          if (
              burnAmount
                  > LiquidationLib.tokensRequiredForTargetHealthFactor(
                      liquidation.threshold + liquidation.maxHealthFactor,
                      mintAmount,
                      tcapPrice,
                      collateralAmount,
                      collateralPrice,
                      liquidation.penalty,
                      assetDecimals
                  )
          ) {
              revert InvalidValue(IVault.ErrorCode.HEALTH_FACTOR_ABOVE_MAXIMUM);
          }
      }
```

```
  function tokensRequiredForTargetHealthFactor(
  // ...
  ) internal pure returns (uint256) {
      uint256 currentHealthFactor = healthFactor(mintAmount, tcapPrice, collateralAmount,
↪  collateralPrice, collateralDecimals);
  // ...
  }
```

- AaveV3Pocket: During a deposit (`_onDeposit()`) Aave is given an allowance to have access exactly to the amount of token being deposited. Consider changing the 'approve() call to give an unlimited allowance to make deposits more gas efficient.

```
  + function initialize() public override initializer {
  +     UNDERLYING_TOKEN.approve(address(POOL), type(uint256).max);
  + }

    /// @dev deposits underlying token into Aave v3, aTokens are deposited into this pocket
    function _onDeposit(uint256 amountUnderlying) internal override returns (uint256 amountOverlying) {
  -     UNDERLYING_TOKEN.approve(address(POOL), amountUnderlying);
        POOL.deposit(address(UNDERLYING_TOKEN), amountUnderlying, address(this), 0);
        return amountUnderlying;
    }
```

- Multicall.sol Mark inline assembly blocks as memoy-safe (Since Solidity 0.8.13), ensuring proper compiler optimization.

```
  - assembly {
  + assembly ("memory-safe") {
```

- BasePocket.sol: Remove the redundant line 65, which loads the user's current balance but does not make any use of ti.

```
    function withdraw(address user, uint256 amountUnderlying, address recipient) external onlyVault
↪   returns (uint256 shares) {
        if (amountUnderlying == type(uint256).max) {
  -         amountUnderlying = _balanceOf(user);
            shares = sharesOf(user);
        } else {
            shares = amountUnderlying * totalShares() / _totalBalance();
        }
```

- TCAPV2.sol: The TCAPv2 ERC-20 contract keeps track of how many tokens have been minted per Vault. During minting, it's impossible for amounts to overflow for a Vault before they overflow the total supply value. During burning, it's impossible for a Vault to burn more tokens than it minted due to the explicit check for this. In both the minting and the burning functions, the addition and subtraction may safely skip the overflow checks automatically added by the Solidity compiler. Consider wrapping these lines within an unchecked block.

```
    function mint(address to, uint256 amount) external onlyRole(VAULT_ROLE) {
        TCAPV2Storage storage $ = _getTCAPV2Storage();
  +     unchecked {
            $._mintedAmounts[msg.sender] += amount;
  +     }
        _mint(to, amount);
        emit Minted(msg.sender, to, amount);
    }

    /// @inheritdoc ITCAPV2
    function burn(address from, uint256 amount) external onlyRole(VAULT_ROLE) {
        TCAPV2Storage storage $ = _getTCAPV2Storage();
        if (amount > $._mintedAmounts[msg.sender]) revert BalanceExceeded(msg.sender);
        _burn(from, amount);
  +     unchecked {
            $._mintedAmounts[msg.sender] -= amount;
  +     }
        emit Burned(msg.sender, from, amount);
    }
```

- Vault.sol: The `depositWithPermit()` function currently immediately copies the contents of the `permit` parameter into memory due to specifying this as data location. This is unnecessary and some gas can be saved avoiding copying by specifying `calldata` as parameter data location.

```
  - function depositWithPermit(uint96 pocketId, uint256 amount, IPermit2.PermitTransferFrom memory
↪   permit, bytes calldata signature)
  + function depositWithPermit(uint96 pocketId, uint256 amount, IPermit2.PermitTransferFrom calldata
↪   permit, bytes calldata signature)
```

Cryptex: None of these findings have a significant impact on the efficiency or security of the protocol. Some of the things pointed out were addressed within the following Pull Requests:

- PR 9.
- PR 10.

Cantina Managed: Fixed.

4.5 Informational

4.5.1 Create Token Integration Checklist

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Like other contracts of Cryptex before, TCAPv2 too is planned to be under control of Community Governance (DAO). This includes the creation of additional Vaults, of which there'll likely be an increasing amount in order to attract liquidity.

To allow the community to verify the Vaults being added, an important aspect is checking whether the token supposed to serve as collateral is properly compatible with the protocol. For this purpose, it's recommendable to provide those who will vote on such proposals a "Token Integration Checklist".

Recommendations: We provide the following recommendations regarding the creation of such a checklist based on our review and understanding of the protocol:

- **Token Standards:** This protocol only deals with underlying tokens of ERC-20 compatible standards. There exist other fungible token standards such as ERC-1155, but these are not supported.
- **Double-Entry-Point Tokens:** i.e. tokens that share the same tracking of balances but have two separate contract addresses from which this balances can be controlled. Typically protocols that have sweeping functions (for rescuing funds) are vulnerable to these since they bypass checks preventing sweeping of underlying funds. Pocket contracts do not appear to be vulnerable to this and using such tokens as underlying should not cause any issues.
- **Token Error Handling:** ERC-20 Tokens historically handle errors in two possible ways, they either revert on errors or they simply return a `false` boolean as a result. With the fixes applied, this protocol correctly handles both cases thanks to usage of solady SafeTransferLib's `safeTransferFrom()`.
- **ERC-20 Optional Decimals:** Within the ERC-20 standard, the existence of a `decimals()` function is optional. This protocol however requires it to be present (`BaseOracleUSD.assetDecimals`) and constant over its lifetime. Assuming the function is implemented by the token, it should return its value as `uint8` type, if not, the value must be below 255.
- **Tokens with Callbacks:** There exist various standard extensions such as ERC-223, ERC-677, ERC-777, etc., as well as custom ERC-20 compatible token implementations that call the sender, receiver, or both, during a token transfer. Furthermore, such implementations may choose to call before or after the token balances were updated. This is especially dangerous since it may allow re-entering the protocol and exploit incomplete state updates.
  - To avoid any issues with such tokens, it's recommended to follow the CEI-pattern. There's currently only one place apparent where this pattern is not being followed: Within Vault.sol the withdrawal of tokens should come after the burning of TCAP debt.

    ```diff
    - pocket.withdraw(user, liquidationReward, msg.sender);
      TCAPV2.burn(msg.sender, burnAmount);
    + pocket.withdraw(user, liquidationReward, msg.sender);
    ```

- **Deflationary/Inflationary or Rebasing Tokens:** There are tokens (such as Aave's aToken) which increase in balance over time, or decrease in balance over time (various algorithmic stable coins), this may cause accounting issues within smart contracts holding them. Pockets should be resistant to issues related to this since they take account of shares, not the actual underlying balances. Whether an underlying token decreases or increases in balance, it would effectively decrease or increase the locked underlying value behind each share.
- **Tokens with Transfer Fees:** There are tokens which may charge a fee for transfers. This fee could be applied on the value being sent, decreasing the amount reaching the receiver, or it could be applied on the sender's remaining balance. Pocket contracts are currently not equipped to handle either of these cases appropriately, and won't be without significant changes to how deposits and withdrawals are handled.
- **Tokens with strict Allowance Handling:** There exist tokens which error when attempting to change an existing token allowance from one non-zero value to another non-zero value. At the moment this is only relevant for the `AaveV3Pocket` contract which should be able to handle this whether the

logic stays as is (approving the exact amount to be deposited) or is changed according to other recommendations within this audit (unlimited approval once in the constructor).

- **Non-Standard Decimals:** Tokens typically have 18 decimals, but some deviate from this, usually towards lower numbers. During this audit we've pointed out a few places where the handling of such non-standard decimals was lacking. Assuming those were all places and they've been fixed appropriately, this protocol should be able to handle such assets.
- **Extreme Scale Deviations:** There may be tokens which have both
  1. Very low decimal values and...
  2. Are very valuable.

  Meaning that the balances when dealing with this token could be so small that rounding errors lead to significant differences in monetary value. It should be checked whether tokens fall into such a category and if potentially so, it should be mathematically verified whether rounding errors are within acceptable ranges.
- **Oracle Integration:** For an asset to be used as collateral within this protocol it must have an active Chainlink Price Feed. Furthermore, the AggregatedChainlinkOracle contract adjusts the price responses to 18 decimals based on the reported `decimals()` of the feed. As a sanity check it should be both verified that:
  1. This decimals value indeed is below 18 as expected by the contract and...
  2. The reported decimals value truthfully matches the actual prices currently reported by the feed.

Consider the consulting the Crytic Checklist (Trail of Bits) for further inspiration.

Cryptex: Acknowledged.

Cantina Managed: Acknowledged.

4.5.2 Upgradeability: Automate verification of storage changes

Severity: Informational

Context: TCAPV2.sol#L12-L16

Description: The TCAP Token, its Vaults and Pockets are upgradeable through the use of Transparent Proxies. For storage management, the contracts specifically make use of the ERC-7201 Namespaced Storage Layout pattern.

While this is different to the previous more common pattern of using `_gaps[]` to ensure that variables may be added to upgradeable contracts at a later time, what remains unchanged is the need to ensure that any changes made to existing variables (ie. reusing existing storage slots for new storage variables) are compatible.

OpenZeppelin provides an Upgrades Plugin for Foundry that automates such compatibility checks between contract upgrades.

Recommendation: Consider setting up CI rules that automatically check changes to namespaced storage for issues.

Cryptex: Acknowledged.

Cantina Managed: Acknowledged.

4.5.3 Various small improvements and other nitpicks

Severity: Informational

Context: AaveV3Pocket.sol#L33, AaveV3Pocket.sol#L4, BasePocket.sol#L108-L116, DeployTCAP.s.sol#L11, DeployTCAP.s.sol#L42, DeployTCAP.s.sol#L5, FeeCalculatorLib.sol#L4, IOracle.sol#L17-L18, LiquidationLib.sol#L4, Vault.sol#L141, Vault.sol#L162-L164, Vault.sol#L247, Vault.sol#L263, Vault.sol#L330

Description: During the code review, a variety of smaller changes with the goal of improving readability and following general best practices were suggested.

Recommendations: Fix those highlighted nitpicks. These are some of the more noteworthy findings:

- Vault.sol: Although there is no apparent issue with specifying zero amounts when interacting with the Vault contract (Deposit, Withdrawal, Minting, or Burning), it's still one of those extreme cases where one could consider "failing early" to ensure nothing weird can happen down the line. Consider explicit checks against 0 amounts that wouldn't have any desirable effect anyway.

```
function deposit(uint96 pocketId, uint256 amount) external returns (uint256 shares) {
+    require(amount > 0);
// ...
```

- Vault.sol: Functions that should still function normally for disabled pockets (such as `withdraw()`) won't call `_getPocket()` and therefore also skip over its implicit sanity check on whether that pocket exists at all. Consider validating `pocketId` parameters of external functions by requiring the pocket's address to be non-zero address.

```
    function withdraw(uint96 pocketId, uint256 amount, address to) external ensureLoanHealthy(msg.sender,
↪  pocketId) returns (uint256 shares) {
       // @audit should be able to withdraw even if pocket is disabled
        IPocket pocket = _getVaultStorage().pockets[pocketId].pocket;
+       require(address(pocket) != address(0x0));
```

- AaveV3Pocket.sol: The `_balanceOf()` function of `AaveV3Pocket.sol` is lacking a check to prevent division-by-zero which the `_balanceOf()` function from `BasePocket.sol` has which is being overriden. Consider adding this check here as well for consistency:

```
    function _balanceOf(address user) internal view override returns (uint256) {
+       uint256 totalShares_ = totalShares();
+       if (totalShares_ == 0) return 0;
-       return sharesOf(user) * OVERLYING_TOKEN.balanceOf(address(this)) / totalShares();
+       return sharesOf(user) * OVERLYING_TOKEN.balanceOf(address(this)) / totalShares_;
    }
```

- BasePocket.sol: In `_balanceOf()` and `_totalBalance()` the balance of `OVERLYING_TOKEN` should be used, instead of `UNDERLYING_TOKEN`, this is because by design the Pocket holds OVERLYING_TOKEN tokens. Although not being an issue, since `BasePocket` has `OVERLYING_TOKEN == UNDERLYING_TOKEN` and `AaveV3Pocket` is already implemented correctly, consider adjusting this for consistency.

```
    function _balanceOf(address user) internal view virtual returns (uint256) {
        uint256 totalShares_ = totalShares();
        if (totalShares_ == 0) return 0;
-       return sharesOf(user) * UNDERLYING_TOKEN.balanceOf(address(this)) / totalShares_;
+       return sharesOf(user) * OVERLYING_TOKEN.balanceOf(address(this)) / totalShares_;
    }

    function _totalBalance() internal view virtual returns (uint256) {
-       return UNDERLYING_TOKEN.balanceOf(address(this));
+       return OVERLYING_TOKEN.balanceOf(address(this));
    }
```

Cryptex: As none of these have any significant impact on the protocol's security, the Cryptex Team addressed some of these findings at their own discretion:

- PR 9.
- PR 11.
- PR 13.

- Commit d1799f6e.
- Commit 3ef6bd16.

Cantina Managed: Fixed.

4.5.4 Review of `TokenExchangeSetIssuer`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: While this audit was mostly centered around the Cryptex's TCAPv2 repository, a single peripheral contract named `TokenExchangeSetIssuer` was also part of the scope, although part of a different repository (Cryptex's crypdex).

Crypdex too is about the creation of Index Tokens, but while TCAPv2 is a synthetic token following the Total Crypto Market Capitalization (divided by 10 billion), this codebase uses `SetTokens` where a single token is in fact backed by a basket (set) of the actual tokens it is indexing.

To mint a SetTokens you therefore have to deposit the correct amounts of each of the underlying tokens, an operation that is rather tedious to execute without automation, which is where the `TokenExchangeSetIssuer` contract in question comes in. The goal is for there to be a User Interface in which you simply specify the amount of SetTokens you'd like to obtain and provide an allowance for a single asset with which all the tokens belonging to the index will be obtained with (under the hood the contract will make swaps on Uniswap and Paraswap).

Being a peripheral contract means that `TokenExchangeSetIssuer` does not have any special access within the system. Rather, it is a helper contract outside of the protocol that anyone could deploy for themselves and it would work just fine. In the normal case, it should never hold any user funds for longer than a single transaction, during which a user is calling it to either buy or sell the necessary underlying tokens for obtaining or discarding some amount of SetTokens.

As such there's practically one critical concern: There may not be any way to make unauthorized `transferFrom()` calls that move funds that `TokenExchangeSetIssuer` has been approved to use. In other words, while the contract should normally never hold funds, it'll very likely end up having allowances to many user's funds, and it must not happen that another use make use of funds that are not their own.

Recommendations: No critical issues in regards to the unauthorized transfer of user-approved funds were found. However, various suggestions around hardening, simplifying and optimizing the code were given, such as

- Upgrading Solidity from 0.6.10 to a newer version, preferably 0.8.26, which is the same version used in TCAPv2. While most of the contracts within the crypdex repository make use of 0.6.10, there shouldn't be much of an issue using a higher version for a peripheral contract that only interacts with crypdex from "outside".
- This upgrade allows making use of the various new language features and bugfixes, but was specifically suggested in order to avoid the use of inline assembly, which is notorious for introducing unexpected behavior.
- The manner by which the contract ensured only swap functions of legitimate exchanges could be called appeared unnecessarily restrictive and inefficient. A change to a simpler whitelist-based implementation, specifically restricting the addresses and function signatures being called, was suggested.
- It was noted that under some circumstances (such as using incorrect swap functions or sending funds by accident) funds could accumulate on the contract. The contract's logic attempted to take account of such "stuck" funds, prevent their usage by other users, and allow their rescue by an administrator. But as these measures can be bypassed, it's recommended to add appropriate warnings for developers where possible to prevent such accumulations in the first place.

Cryptex: The Cryptex team made several changes to `TokenExchangeSetIssuer` on PR 6, addressing many of the things pointed out during the review. The PR is split into 3 commits:

- Summary of Commit 1:
  - Reverted to code from an earlier commit, enhancing contract flexibility to allow more exchanges to be whitelisted and invoked in the future.

  - Updated the whitelisting logic to create unique identifiers using the target contract address and the 4-byte function selector.
  - Added warnings advising users not to send tokens or ETH directly to the contract.
  - Included a caution that any residual dust left from buying or selling will not be refunded.
- Summary of Commit 2:
  - Added OpenZeppelin v5.0.2 to support Solidity v0.8.26, aliasing it as `@openzeppelin-contracts-5` to differentiate from the version used by core contracts.
- Summary of commit 3:
- Introduced new interfaces compatible with Solidity v0.8.26.
- Upgraded the Solidity version to v0.8.26 for the `TokenExchangeIssuer`.
- Got rid of the assembly code introduced in commit 1 for converting `bytes` to `bytesN`.
- Leveraged the updated Solidity version to replace require messages with custom error handling.
- Emitted events for the following functions: `whitelistFunctions`, `revokeWhitelistedFunctions`, `addSetTokenIssuanceModules`, and `removeSetTokenIssuanceModules`.

Suggestions that were not implemented:

- Couldn't find `ReentrancyGuardTransient` in the latest OpenZeppelin version v5.0.2. It only appears in the master branch, so I was unable to replace `ReentrancyGuard` with `ReentrancyGuardTransient`.

Cantina Managed: Fixed.

4.5.5 Deposit storage struct has unused members

Severity: Informational

Context: Vault.sol#L24-L27

Description: The Vault contract makes extensive use of `struct` types for managing its storage. It appears that the struct used for tracking data on user deposits has attributes that aren't actually used anywhere within the Vault's logic.

Recommendation: Remove the dead code:

```
  struct Deposit {
-     address user;
-     uint96 pocketId;
-     bool enabled;
      uint256 mintAmount;
      uint256 feeIndex;
      uint256 accruedInterest;
  }
```

Cryptex: Addressed in PR 11.

Cantina Managed: The Cryptex Team has addressed this issue within PR 11.

4.5.6 Use `safeTransfer()` instead of `transfer()` in `BasePocket`

Severity: Informational

Context: BasePocket.sol#L105

Description: While walking through the code base during the audit's kickoff meeting, the Cryptex team noticed that one of the token transfers did not make use of `safeTransfer()`. This finding serves as a reminder for this to be fixed.

Recommendation: The issue was specifically identified within the BasePocket.sol file:

```
- UNDERLYING_TOKEN.transfer(recipient, amountUnderlying);
+ address(UNDERLYING_TOKEN).safeTransfer(recipient, amountUnderlying);
```

Cryptex: Addressed in PR 8.

Cantina Managed: The Cryptex Team has addressed this issue within PR 8.