Cryptex Security Review

Pashov Audit Group

Conducted by: DadeKuma, DanOgurtsov, ZanyBonzy

September 30th - October 4th

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **cryptexfinance/tcapv2.0** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About TCAP

The Total Market Cap Token (TCAP) by Cryptex Finance is a crypto index token that provides investors with exposure to the entire cryptocurrency market's value through a single synthetic asset.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - f0168f3fe66c1fba4fd70ffdcdc87287e8f0cb6a

fixes review commit hash - 71ee4a810febe2cb5fcbf2102076b44525005289

Scope

The following smart contracts were in scope of the audit:

- `TCAPV2`
- `Vault`
- `Constants`
- `FeeCalculatorLib`
- `LiquidationLib`
- `Multicall`
- `AggregatedChainlinkOracle`
- `BaseOracleUSD`
- `TCAPTargetOracle`
- `AaveV3Pocket`
- `BasePocket`

7. Executive Summary

Over the course of the security review, DadeKuma, DanOgurtsov, ZanyBonzy engaged with Cryptex to review TCAP. In this period of time a total of **12** issues were uncovered.

Protocol Summary

Protocol Name	TCAP
Repository	https://github.com/cryptexfinance/tcapv2.0
Date	September 30th - October 4th
Protocol Type	Synthetic asset

Findings Count

Severity	**Amount**
Medium	2
Low	10
Total Findings	**12**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Add a form of access control to takeFee	Medium	Resolved
[M-02]	Circuit breakers are not considered when processing Chainlink's answer	Medium	Resolved
[L-01]	No check for Arbitrum Sequencer when handling prices	Low	Acknowledged
[L-02]	Use of a general staleness time is not advisable	Low	Resolved
[L-03]	Withdrawal may sometimes fail if fee taken is high enough	Low	Acknowledged
[L-04]	Users can entirely skip paying fees if there is no feeRecipient	Low	Resolved
[L-05]	Full liquidation can be drawn out or temporarily prevented	Low	Resolved
[L-06]	Users that mint zero shares will lose their funds	Low	Resolved
[L-07]	Liquidators might get a reward that is less than expected	Low	Acknowledged
[L-08]	Excessive loans to trigger a liquidation	Low	Acknowledged
[L-09]	Pocket inflation attack	Low	Acknowledged
[L-10]	Loans that use an AAVE pocket may become non-liquidatable	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] Add a form of access control to `takeFee`

Severity

Impact: Medium

Likelihood: Medium

Description

`takeFee()` function is available for everyone. Imagine depositors A, B, C, each deposited 10 ETH, and holding 10 shares each, and they have the same debt. Anyone can call `takeFee()` for A and B, decreasing their shares gradually to 9.99. As a result, depositor C has more shares.

```
function takeFee(address user, uint96 pocketId) external {
        IPocket pocket = _getVaultStorage().pockets[pocketId].pocket;
        if (address(pocket) == address(0)) revert InvalidValue
          (IVault.ErrorCode.INVALID_POCKET);
        _takeFee(pocket, user, pocketId);
    }
```

Recommendation

Consider limiting the function to a protected role and/or allowing a user to only take fee for himself.

[M-02] Circuit breakers are not considered when processing Chainlink's answer

Severity

Impact: High

Likelihood: Low

Description

Every Chainlink feed has a minimum and maximum price. However, due to the circuit breaker, if an asset's price moves outside these limits, the provided answer will still be capped.

This can lead to an incorrect price if the actual price falls below the aggregator's `minAnswer`, as Chainlink will continue providing the capped value instead of the true one.

The Chainlink documentation notes that "On most data feeds, these values are no longer used and they do not prevent your application from reading the most recent answer.". However, this is not the case on Arbitrum, as for most data feeds (including ETH and most stablecoins), these values are indeed used, for example, the ETH/USD aggregator: link

Recommendations

Consider checking that the price is always between the aggregator's `minAnswer` and `maxAnswer`:

```
function latestPrice
      (bool checkStaleness) public view virtual override returns (uint256) {
        ...
-       assert(answer > 0);
+       require(answer > MIN_ANSWER && answer < MAX_ANSWER);
        ...
    }
```

8.2. Low Findings

[L-01] No check for Arbitrum Sequencer when handling prices

Contracts will be deployed on Arbitrum but AggregatedChainlinkOracle.sol doesn't check if Arbitrum Sequencer is down. If it goes down, oracle data will not be kept up to date, and thus could become stale. However, users are able to continue to interact with the protocol directly through the L1 optimistic rollup contract. While a number of the protocol operations actually allow for stale pricing, some don't, e.g. minting TCAPV2.

As a result, during the sequencer downtime period, users whose position might otherwise be unhealthy at the actual price may be able to continue minting because stale prices are in use or vice versa.

Recommend following chainlink's example code to set up checks for the sequencer.

[L-02] Use of a general staleness time is not advisable

Staleness is checked while minting by ensuring that the last update time is not more than a day. However, not all tokens have a 1-day stale period. Some are as short as one hour. The implementation will allow stale prices to be accepted for these tokens.

```
if (checkStaleness) {
            if (updatedAt < block.timestamp - 1 days) {
                revert StaleOracle();
            }
        }
```

Consider using a mapping that would record the heartbeat parameter of each token to determine the stale period.

[L-03] Withdrawal may sometimes fail if fee taken is high enough

When users withdraw, a fee is first taken:

```
function withdraw(uint96 pocketId, uint256 amount, address to) external {
        // ...
>>      _takeFee(pocket, msg.sender, pocketId);
>>      shares = pocket.withdraw(msg.sender, amount, to);
    }
```

As can be seen, `_takeFee` withdraws the interest from the user's pocket.

```
function _takeFee(IPocket pocket, address user, uint96 pocketId) internal {
//...
            pocket.withdraw(user, interest, feeRecipient_);
//...
```

The withdrawn interest then reduces the total number of shares that the user has.

```
function withdraw(
      addressuser,
      uint256amountUnderlying,
      addressrecipient
    ) external onlyVault returns (uint256 shares
///...
        $.sharesOf[user] -= shares;
        $.totalShares -= shares;
//...
```

After this process, the `amount` entered in by the user is now attempted to be withdrawn.

```
function withdraw(uint96 pocketId, uint256 amount, address to) external {
        // ...
        shares = pocket.withdraw(msg.sender, amount, to);
    }
```

If the fee is high enough, the subsequent shares calculated user from the user's inputted amount will now be more than the user's `shareOf` which has now been reduced by the taken fee which the cause the withdrawal to fail.

To prove this, we add the test below to Vault.t.sol and run it with `forge test --mt test_mayNotBeAbleToWithdraw -vvv`

Copy and add the test below to Vault.t.sol, and run it. We can see that the 100 ether deposited by the user is fully withdrawn, without regard for the fee accumulated during the period.

```
function test_userMayPayNoFees(uint32 timestamp) public {
        address user = makeAddr("user");
        uint256 amount = 100 ether;
        uint256 mintAmount = amount * 1e10 / 10;
        uint256 burnAmount = mintAmount;
        deposit(user, amount);
        vm.prank(user);
        vault.mint(pocketId, mintAmount);
        timestamp = uint32(bound(timestamp, block.timestamp + 10000, type
          (uint32).max));
        vm.warp(timestamp);
        vm.prank(user);
        vault.burn(pocketId, burnAmount);
        // We set fee recipeint to 0
        vault.updateFeeRecipient(address(0));
        uint256 outstandingInterest = vault.outstandingInterestOf
          (user, pocketId);
        assertGt(outstandingInterest, 0);
        vm.prank(user);
        //User can safely withdraw all of his token
        vault.withdraw(pocketId, 100 ether, user);
    }
```

I'd recommend either preventing withdrawals while there's no fee recipient or withdrawing the amount minus the expected fee to the user. The fee can then be taken later when the recipient is set.

[L-05] Full liquidation can be drawn out or temporarily prevented

`liquidate` reverts if the liquidator's `burnAmount` is more than the user's `mintAmount`. As a result, full liquidation can be made difficult by the user burning a small amount of his TCAPV2, which is enough to reduce his `mintAmount` but may not be enough to raise his health factor. This can prove to be frustrating for liquidators, especially in cases where full liquidation is more profitable than partial.

Add the test below to Vault.t.sol and run it. It shows that a user can cause liquidation to fail by burning 1 wei of his TCAPV2 without actually making his loan any healthier.

```
function test_frontrunLiquidation(address user, uint256 amount) public {
        vm.assume(user != address(0) && user != address(vaultProxyAdmin));
        uint256 depositAmount = deposit(user, amount, 1, 1e28);

        vm.startPrank(user);
        uint256 mintAmount = bound(amount, 1, depositAmount * 1e10);
        vault.mint(pocketId, mintAmount);
        vm.stopPrank();

        uint256 collateralValue = vault.collateralValueOfUser(user, pocketId);
        uint256 mintValue = vault.mintedValueOf(mintAmount);
        uint256 multiplier = mintValue * 10_000 / (collateralValue);
        vm.assume(multiplier > 1);
        feed.setMultiplier(multiplier - 1);

        vm.startPrank(user);
        vault.burn(pocketId, 1);
        vm.stopPrank();

        //We attempt to mint 1 wei to show that user's loan is still not healthy
        vm.expectRevert();
        vm.startPrank(user);
        vault.mint(pocketId, 1);
        vm.stopPrank();

        tCAPV2.mint(address(this), mintAmount);
        vault.liquidate(user, pocketId, mintAmount);
    }
```

The test should fail with the error code 10 which is equivalent to `INVALID_BURN_AMOUNT`

```
|   |   |   └─ ← [Return] 958464810053752764273512126 3 [9.584e27]
|   |   |   └─ ← [Revert] InvalidValue(10)
|   |   └─ ← [Revert] InvalidValue(10)
|   └─ ← [Revert] InvalidValue(10)
```

Recommend not reverting if `burnAmount` > `mintAmount`, rather set `burnAmount` to `mintAmount` instead.

```
// ...
        uint256 mintAmount = mintedAmountOf(user, pocketId);
-       if (burnAmount > mintAmount) revert InvalidValue
- (IVault.ErrorCode.INVALID_BURN_AMOUNT);
+       if (burnAmount > mintAmount) {
+       burnAmount = mintAmount;
+      }
        uint256 tcapPrice = TCAPV2.latestPrice();
// ...
```

[L-06] Users that mint zero shares will lose their funds

When a user deposit their funds, the following function will be called:

```
function registerDeposit(
      addressuser,
      uint256amountUnderlying
    ) external onlyVault returns (uint256 shares
        uint256 amountOverlying = _onDeposit(amountUnderlying);
        uint256 totalShares_ = totalShares();
        if (totalShares_ > 0) {
            shares = (totalShares_ * amountOverlying) /
              (OVERLYING_TOKEN.balanceOf(address(this)) - amountOverlying);
        } else {
            shares = amountOverlying * Constants.DECIMAL_OFFSET;
        }
        BasePocketStorage storage $ = _getBasePocketStorage();
        $.totalShares += shares;
        $.sharesOf[user] += shares;
        // @audit should not happen, prevent overflow when calculating balance
        assert($.sharesOf[user] < 1e41);
        emit Deposit(user, amountUnderlying, amountOverlying, shares);
    }
```

If `totalShares_ > 0` and the denominator is higher than the numerator, the user will deposit their funds but they will receive zero shares in return, so consider reverting when `shares == 0`.

[L-07] Liquidators might get a reward that is less than expected

In `Vault`, the `liquidationReward` depends on the current price of TCAP. A liquidator will burn `burnAmount` of their tokens, but if the price of TCAP is currently volatile, they might get fewer rewards than expected.

Consider adding a `minRewardAmount` for a liquidation, and revert the transaction if the reward is less than this value.

[L-08] Excessive loans to trigger a liquidation

In `Vault`, users can get a loan up to the liquidation threshold:

```
if (_healthFactor(user, pocketId, checkStaleness) < liquidationParams
   ().threshold) revert LoanNotHealthy();
```

Liquidation can be triggered if the health factor is at least equal to the previous threshold:

```
function liquidate(
      addressuser,
      uint96pocketId,
      uint256burnAmount
    ) external returns (uint256 liquidationReward
        ...
        IVault.LiquidationParams memory liquidation = liquidationParams();
        uint256 healthFactor_ = LiquidationLib.healthFactor(
          mintAmount,
          tcapPrice,
          collateralAmount,
          collateralPrice,
          COLLATERAL_DECIMALS
        );
->      if (healthFactor_ >= liquidation.threshold) revert LoanHealthy();
        ...
    }
```

This is an issue because the difference is only 1 wei, making it highly likely that a maximized loan will be liquidated in the next block.

Consider reducing the threshold at which is possible to increase the loan amount to avoid this scenario.

[L-09] Pocket inflation attack

To kick-start the attack, the malicious user will often usually mint the smallest possible amount of shares (e.g., 1 wei) and then donate significant assets to the vault to inflate the number of assets per share. Subsequently, it will cause a rounding error when other users deposit. link

In the case of Pockets, the attacker will have to donate `OVERLYING_TOKEN`.

```
function registerDeposit(
      addressuser,
      uint256amountUnderlying
    ) external onlyVault returns (uint256 shares
        uint256 amountOverlying = _onDeposit(amountUnderlying);
        uint256 totalShares_ = totalShares();
        if (totalShares_ > 0) {
            shares = (totalShares_ * amountOverlying) /
              (OVERLYING_TOKEN.balanceOf(address(this)) - amountOverlying);
        } else {
            shares = amountOverlying * Constants.DECIMAL_OFFSET;
        }
        BasePocketStorage storage $ = _getBasePocketStorage();
        $.totalShares += shares;
        $.sharesOf[user] += shares;
        // @audit should not happen, prevent overflow when calculating balance
        assert($.sharesOf[user] < 1e41);
        emit Deposit(user, amountUnderlying, amountOverlying, shares);
    }
```

The first depositor will receive `shares = amountOverlying * Constants.DECIMAL_OFFSET`. But this protection is not enough as this depositor still can withdraw collateral to leave only 1 wei of shares.

Test to reproduce the attack: Gist

Consider using virtual shares as explained in the link above.

[L-10] Loans that use an AAVE pocket may become non-liquidatable

In `AaveV3Pocket`, the function will revert on withdrawal when the supply available on Aave is not sufficient to cover the entire amount:

```
function _onWithdraw(
      uint256amountOverlying,
      addressrecipient
    ) internal override returns (uint256 amountUnderlying
        if (amountOverlying == 0) return 0;
        uint256 amountWithdrawn = POOL.withdraw(address
          (UNDERLYING_TOKEN), amountOverlying, recipient);
        // https://github.com/code-423n4/2022-06-connext-findings/issues/181
->      assert(amountWithdrawn == amountOverlying);
        return amountOverlying;
    }
```

This can occur during a liquidity shortage on Aave, and it's meant to be a protection for users when they withdraw their collateral.

The issue is that this may also happen during a liquidation, as it would cause a revert. In `Vault.liquidate`, `_takeFee` will withdraw the accrued interest, and this may block the transaction, resulting in the accumulation of bad debt for the protocol:

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

As the conversion between underlying and overlying is always 1:1, sending users the overlying token would guarantee successful liquidations always.

The alternative option:

In `AaveV3Pocket`, consider avoiding a revert when `amountOverlying` is not equal to `amountWithdrawn`. Instead, consider using a pull approach: store the difference and ensure that the recipient can withdraw it.