Flying Tulip, Sherlock Audit

Version 1.0

Tanu Gupta

Jan 17, 2026

Flying Tulip, Sherlock

Tanu Gupta

Jan 17, 2026

Prepared by: Tanu Gupta Lead Security Researcher: - Tanu Gupta

Table of Contents

    - Issues found

- Findings
- Medium
    - [M-1] CircuitBreaker Owner can exceed mainBuffer cap via emergencyOverride without time-based updates leading to governance risk
    - [M-2] Accounting corruption in amountInvested mapping of ftACL when mixing `proofAmount = 0` and `proofAmount > 0` during investment by whitelisted users
    - [M-3] Malicious actors / MEV bots can monopolize the global cap of investment leading to DOS
- Low
    - [L-1] Silent truncation when casting to uint64 can cause acceptance of msig update sooner than intended
    - [L-2] CircuitBreaker main buffer cap uses stale TVL, creating design trade-off between flash loan protection and rate limiting accuracy
    - [L-3] CircuitBreaker rate limit based on total value (including yield) creates mismatch with actual withdrawable capacity
- Informational
    - [I-1] `yieldWrapper::withdrawQueued()` and `yieldWrapper::claimQueued()` will always revert when called with strategies that don't implement `IStrategyWithQueue`

Protocol Summary

Overview

Flying Tulip PUT (pFT) is a cash-secured put options protocol that enables users to deposit collateral (e.g., USDC, USDT, wrapped native tokens) in exchange for PUT NFTs representing their positions. The protocol's core value proposition is **principal protection with yield generation**: all deposited collateral is deployed to yield-generating strategies, where the principal remains protected and yield accrues to the protocol treasury.

Core Architecture

The protocol consists of four main components:

1. **PutManager** (`contracts/PutManager.sol`): The primary entry point that orchestrates the entire protocol lifecycle. It manages the public offering phase, handles user investments, maintains a collateral registry, tracks FT token allocation, and provides post-sale exit mechanisms. The contract is upgradeable via UUPS proxy pattern and implements access control through multiple roles (msig, configurator).
2. **pFT** (`contracts/pFT.sol`): An ERC721Enumerable NFT contract representing PUT positions. Each investment mints a unique PUT NFT to the investor, which encodes the position's collateral amount, strike price, and token type. Only PutManager can mint, burn, or modify these positions.
3. **ftYieldWrapper** (`contracts/ftYieldWrapper.sol`): An ERC20 wrapper contract that manages collateral deployment to yield strategies. It maintains a 1:1 share ratio with principal deposits, meaning each wrapper share represents exactly one unit of underlying collateral. The wrapper handles strategy allocation, yield claiming, and provides two withdrawal mechanisms: exact underlying withdrawal and in-kind position token withdrawal.
4. **Strategy Adapters** (`contracts/strategies/*`): Modular strategy contracts (e.g., AaveStrategy, StEthStrategy, EthenaSUSDeStrategy) that interface with external DeFi protocols. Strategies maintain 1:1 principal shares and isolate yield in position tokens, which are periodically claimed and sent to the treasury.

Protocol Lifecycle

Public Offering Phase

During the offering phase (`saleEnabled = true`):

- **Investment Flow**: Users call `invest(token, amount, proofAmount, proofWL)` to deposit collateral. The protocol:
  - Pulls collateral from the user to PutManager
  - Deposits collateral into the appropriate `ftYieldWrapper`, which mints wrapper shares 1:1 to PutManager
  - Prices the investment using `IFlyingTulipOracle` to determine FT token allocation based on `ftPerUSD()` and asset price
  - Mints a PUT NFT (pFT) to the investor representing their position
  - Tracks FT allocation against `ftOfferingSupply` to prevent overallocation
  - Optionally enforces whitelist caps via `ftACL` if configured
- **Collateral Management**: The protocol supports multiple collateral types, each with its own wrapper vault. Collateral must be whitelisted by the msig via `addAcceptedCollateral(token, vault)`, which validates oracle pricing and token decimals.

- **Sale Control**: The configurator role manages FT liquidity via `addFTLiquidity()`, controls sale state via `setSaleEnabled()`, and can set per-token collateral caps during the offering.

Post-Offering Phase

After `enableTransferable()` is called, positions become transferable and users can exit via three mechanisms:

1. **withdrawFT(id, amount)**: Burns FT tokens from the position and returns them to the owner. The corresponding collateral is tracked in `capitalDivesting[token]` for later withdrawal by the msig. This effectively invalidates the put option.
2. **divest(id, amountFT)**: Burns FT tokens and withdraws the exact underlying collateral amount to the owner via the wrapper. This executes the put option, returning principal to the user.
3. **divestUnderlying(id, amountFT)**: Burns FT tokens and returns position tokens (in-kind) directly to the owner. This allows users to receive strategy position tokens instead of underlying collateral.

Key Mechanisms

Pricing

The protocol uses a single source of truth for pricing: `IFlyingTulipOracle`, which provides: - `getAssetPrice(token)`: Asset price in 1e8 scale - `ftPerUSD()`: FT tokens per USD in 1e8 scale

Conversion helpers in PutManager: - `ftFromCollateral()`: Calculates FT allocation for a given collateral amount - `collateralFromFT()`: Calculates exact collateral required for a given FT amount

Yield Generation and Principal Protection

The protocol enforces strict separation between principal and yield:

- **Principal Protection**: Wrapper shares maintain a 1:1 ratio with deposited principal. Strategies are designed to preserve principal value, with yield accumulating separately in position tokens.
- **Yield Claiming**: Authorized roles (`yieldClaimer`, `subYieldClaimer`) can call `claimYield(strategy)` or `claimYields()` to move surplus yield to the treasury. Idle yield can be swept via `sweepIdleYield()`.
- **Withdrawal Mechanics**: The wrapper implements intelligent withdrawal logic:

  - Uses idle balance first
  - Greedily drains the most liquid strategies (with try/catch on strategy views)
  - Reverts on shortfall to protect principal
  - Burns wrapper shares 1:1 with withdrawn amount

Security Features

- **Circuit Breaker**: Implements ERC-7265-inspired rate limiting with dual buffers (main buffer for time-based replenishment, elastic buffer for deposit tracking) to prevent flash loan DoS attacks and large withdrawal exploits.
- **Reentrancy Protection**: Uses OpenZeppelin's `ReentrancyGuardTransient` across critical functions.
- **Pausability**: PutManager can be paused by msig to halt new investments during emergencies.
- **Access Control**: Multi-role system with timelock delays for critical operations (msig rotation, strategy additions).

Roles and Permissions

- **msig**: Protocol governance multisig. Can pause/unpause, rotate roles, set oracle/ACL, list collateral/vaults, withdraw divested capital, and upgrade contracts.
- **configurator**: Operations multisig. Controls sale flags, FT liquidity, collateral caps, and handles post-sale remainder.
- **yieldClaimer/strategyManager**: Manage strategy deployment, maintenance, and yield claiming.
- **treasury**: Receives all protocol yield.

The protocol is designed to be deployed on multiple chains (Ethereum, Base, BSC, Avalanche, Sonic) with chain-specific token configurations.

Disclaimer

I, Tanu Gupta, make all efforts to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by me is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

I've used the Shelock severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

The audit was conducted on the following contracts: - `contracts/PutManager.sol` - `contracts/pFT.sol` - `contracts/ftYieldWrapper.sol` - `contracts/strategies/AaveStrategy.sol` - `contracts/interfaces/*` - `contracts/cb/CircuitBreaker.sol` - `contracts/FlyingTulipOracle.sol` - `contracts/ftACL.sol`

Roles

The audit was conducted on the following roles: - `msig` - `configurator` - `yieldClaimer` - `strategyManager` - `treasury`

Executive Summary

The audit is conducted corresponding to the following code repository. The audit duration was Jan 5, 2026 to Jan 17, 2026.

Issues found

Severity	Number of issues found
High	0
Medium	3
Low	3
Info	1
Gas	0
Total	7

Findings

Medium

[M-1] CircuitBreaker Owner can exceed mainBuffer cap via emergencyOverride without time-based updates leading to governance risk

Description The `emergencyOverride()` function in CircuitBreaker.sol lines 364-370 has several issues:

- It does not call `_updateBuffers()` first, missing natural replenishment to mainBuffer and decay to elaticBuffer while leaving lastUpdate to stale value even after updating mainBuffer
- It is adding amount to mainBuffer which has a cap instead of elasticBuffer which is temporary and has no cap
- It lacks a currentTvl parameter needed for `_updateBuffers()` Without any bounds, owner can set the mainBuffer to any amount they wish. If set to a higher value this can enable large withdrawals, hence defying the purpose of circuit breaker.

Root Cause: In CircuitBreaker.sol, there are some missing checks on amount that's added to mainBuffer

- There is no check to ensure that after addition mainBuffer remains within the limits of cap
- Natural replenishment from time passage is lost to the new mainBuffer value

```
1  function emergencyOverride(address asset, uint256 amount) external
      onlyOwner {
2          LimiterState storage state = assetState[asset];
3
```

```
4           // Add the amount to main buffer to allow this specific
                withdrawal
5           state.mainBuffer = (uint256(state.mainBuffer) + amount).
                toUint96();
6           emit EmergencyOverride(asset, amount);
7       }
```

Internal Pre-conditions:

1. The Owner deploys the circuitBreaker contract
2. StrategyManager of yieldWrapper for token USDC, calls setCircuitBreaker() to set the address of circuit Breaker

Attack Path:

A user can exploit the circuit breaker by repeatedly calling emergencyOverride to increase the main buffer beyond its intended cap. This allows the user to bypass the rate limit and withdraw large amounts of funds.

1. Initially investor deposits 500MUSDC, with circuit breaker cap set to 5% of TVL i.e 25M USDC
2. Investor tries to withdraw 50M USDC, exceeding the cap and fails
3. Owner modifies the main buffer and adds 50M to mainBuffer, causing main buffer to exceed cap after override
4. Owner modifies the main buffer and adds another 50M to mainBuffer, totaling to 100M USDC
5. Owner adds another 100M to mainBuffer, totaling to 200MUSDC
6. Now when investor tries to withdraw a large amount, this gets allowed by the owner. This is clear scenario of centralization risk.

Impact

1. Buffer state becomes inconsistent with time-based replenishment
2. Stale Timestamps: lastUpdate doesn't reflect the actual last update time
3. mainBuffer can exceed its intended cap
4. emergencyOverride introduces potential centralization risk and may favor certain investors.
5. Owner can temporarily bypass rate limits by coordinating emergencyOverride + withdrawal in the same block

Proof of Concepts Paste the following code in CircuitBreakerIntegration.t.sol to see the results

```
1       function test_emergency_override_Fails_invariant() external {
2           Context memory ctx = _deployFixture();
3           _addCollateralAndLiquidity(ctx);
4           vm.prank(ctx.circuitBreaker.owner());
5           ctx.circuitBreaker.addProtectedContract(address(ctx.wrapper));
6
```

```
7           // Investor deposits 500M USDC
8           _investPosition(ctx);
9           uint256 tvl = ctx.wrapper.valueOfCapital();
10          console2.log("Initial TVL: ", tvl);
11
12          // Get circuit breaker configuration
13          (uint64 maxWad, , ) = ctx.circuitBreaker.config();
14          uint256 cap = (tvl * maxWad) / 1e18; // 5% of 500M = 25M
15          console2.log("Circuit Breaker Cap (5% of TVL): ", cap);
16
17          // Warp time to deplete both buffers
18          vm.warp(block.timestamp + CB_ELASTIC_WINDOW + CB_MAIN_WINDOW +
                1);
19
20
21          // Check initial buffer state (should be depleted)
22          (uint256 mainBufferBefore, uint256 elasticBufferBefore,) =
23              ctx.circuitBreaker.getRawAssetState(address(ctx.usdc));
24
25          console2.log("Main Buffer Before Override: ", mainBufferBefore)
                ;
26          console2.log("Elastic Buffer Before Override: ",
                elasticBufferBefore);
27
28          uint256 totalCapacityBefore = ctx.circuitBreaker.
                withdrawalCapacity(address(ctx.usdc), tvl);
29          console2.log("Total Withdrawal Capacity Before: ",
                totalCapacityBefore);
30
31          // Attempt a large withdrawal that should be rate-limited (50M
                = 10% of TVL, exceeds 5% cap)
32          uint256 largeWithdrawalAmount = 50_000_000 * 1e6; // 50M USDC
33          vm.prank(address(ctx.wrapper));
34          (bool allowedBefore, uint256 availableBefore) =
35              ctx.circuitBreaker.checkAndRecordOutflow(address(ctx.usdc),
                    largeWithdrawalAmount, tvl);
36
37          console2.log("Large Withdrawal (50M) Allowed Before Override: "
                , allowedBefore);
38          console2.log("Available Capacity Before: ", availableBefore);
39          assertFalse(allowedBefore, "Large withdrawal should be rate-
                limited before override");
40
41          // ===== VULNERABILITY: Owner repeatedly calls
                emergencyOverride =====
42
43          // First override: Add 50M to mainBuffer
44          vm.prank(ctx.circuitBreaker.owner());
45          ctx.circuitBreaker.emergencyOverride(address(ctx.usdc), 50
                _000_000 * 1e6);
46
```

```
47          (uint256 mainBufferAfter1,,) = ctx.circuitBreaker.
                getRawAssetState(address(ctx.usdc));
48          console2.log("Main Buffer After 1st Override (50M): ",
                mainBufferAfter1);
49          assertGt(mainBufferAfter1, cap, "Main buffer should exceed cap
                after first override");
50
51          // Second override: Add another 50M (total 100M)
52          vm.prank(ctx.circuitBreaker.owner());
53          ctx.circuitBreaker.emergencyOverride(address(ctx.usdc), 50
                _000_000 * 1e6);
54
55          (uint256 mainBufferAfter2,,) = ctx.circuitBreaker.
                getRawAssetState(address(ctx.usdc));
56          console2.log("Main Buffer After 2nd Override (100M total): ",
                mainBufferAfter2);
57          assertGt(mainBufferAfter2, cap * 4, "Main buffer should be 4x
                the cap");
58
59          // Third override: Add another 100M (total 200M = 8x the cap!)
60          vm.prank(ctx.circuitBreaker.owner());
61          ctx.circuitBreaker.emergencyOverride(address(ctx.usdc), 100
                _000_000 * 1e6);
62
63          (uint256 mainBufferAfter3,,) = ctx.circuitBreaker.
                getRawAssetState(address(ctx.usdc));
64          console2.log("Main Buffer After 3rd Override (200M total): ",
                mainBufferAfter3);
65          assertGt(mainBufferAfter3, cap * 8, "Main buffer should be 8x
                the cap");
66
67          // ===== DEMONSTRATE BYPASS: Large withdrawal now allowed =====
68          uint256 totalCapacityAfter = ctx.circuitBreaker.
                withdrawalCapacity(address(ctx.usdc), tvl);
69          console2.log("Total Withdrawal Capacity After Overrides: ",
                totalCapacityAfter);
70
71          // Now the large withdrawal should be allowed (even though it's
                 10% of TVL, exceeding 5% rate limit)
72          vm.prank(address(ctx.wrapper));
73          (bool allowedAfter, uint256 availableAfter) =
74              ctx.circuitBreaker.checkAndRecordOutflow(address(ctx.usdc),
                    largeWithdrawalAmount, tvl);
75
76          console2.log("Large Withdrawal (50M) Allowed After Override: ",
                 allowedAfter);
77          console2.log("Available Capacity After: ", availableAfter);
78          assertTrue(allowedAfter, "Large withdrawal should be allowed
                after emergency overrides");
79          assertGe(availableAfter, largeWithdrawalAmount, "Available
                capacity should exceed withdrawal amount");
```

```
80      }
```

Logs:

```
1   Initial TVL:  500000000000000
2   Circuit Breaker Cap (5% of TVL):  25000000000000
3   Main Buffer Before Override:  0
4   Elastic Buffer Before Override:  500000000000000
5   Total Withdrawal Capacity Before:  25000000000000
6   Large Withdrawal (50M) Allowed Before Override:  false
7   Available Capacity Before:  25000000000000
8   Main Buffer After 1st Override (50M):  75000000000000
9   Main Buffer After 2nd Override (100M total):  125000000000000
10  Main Buffer After 3rd Override (200M total):  225000000000000
11  Total Withdrawal Capacity After Overrides:  25000000000000
12  Large Withdrawal (50M) Allowed After Override:  true
13  Available Capacity After:  225000000000000
```

Recommended mitigation

1. Add currentTvl parameter and call `_updateBuffers()` first

```
function emergencyOverride(address asset, uint256 amount, uint256
    currentTvl) external onlyOwner {
     LimiterState storage state = assetState[asset];
     // Update buffers first to get natural replenishment
     _updateBuffers(asset, state, currentTvl);
     // Add to elasticBuffer (temporary, no cap) instead of mainBuffer
     state.elasticBuffer += uint96(amount);
     emit EmergencyOverride(asset, amount);
}
```

2. Use `elasticBuffer` instead of `mainBuffer`: Emergency overrides should be temporary, so they belong in elasticBuffer, which decays over time.

[M-2] Accounting corruption in amountInvested mapping of ftACL when mixing `proofAmount = 0` and `proofAmount > 0` during investment by whitelisted users

Description

The `_invest` function in PutManager.sol has inconsistent handling of `proofAmount = 0` (unlimited) vs `proofAmount > 0` (capped) proofs, which can lead to accounting corruption in the amountInvested mapping in ftACL.sol at line 86.

```
function invest(
        address account,
        address token,
```

```
        uint256 amount,
        uint256 proofAmount
    )
        external
        onlyPutManager
    {
        uint256 newInvestedAmount = amountInvested[account][token] +
            amount;
        if (newInvestedAmount > proofAmount) {
            revert ftACLCapReached();
        }

        amountInvested[account][token] = newInvestedAmount;
    }
```

Root Cause:

When `proofAmount = 0`, the per-user cap tracking is completely bypassed:

```
if (address(ftACL) != address(0) && proofAmount != 0) {
    ftACL.invest(msg.sender, token, amount, proofAmount);
}
```

- Causing users with `proofAmount = 0` can invest unlimited amounts(capped to global cap) without any tracking
- Users with `proofAmount > 0` have their investments tracked in `ftACL.amountInvested[user][token]`
- If a user first invests with `proofAmount > 0` (tracked), then later invests with `proofAmount = 0` (not tracked), the amountInvested mapping becomes stale and incorrect

Internal Pre-conditions:

- Alice has some USDC to invest
- Alice is among the whitelisted users to participate in initial offerings
- Alive has two proofs
  - ProofA = (user, USDC, 0) with proofAmount = 0
  - ProofB = (user, USDC, 10000) with proofAmount = 10000 (capped)
- In fact, alice doesn't even need to have two proofs, one proof with non-zero proof amount is enough to show the inconsistency in accounting
- Sale must be enabled for users to invest

Attack Path:

- Alice invests 5,000 USDC using Proof A
- Amount invested remained at 0 => acl.amountInvested(alice, address(usdc)) == 0

- Alice invests another 7000 using Proof B => acl.amountInvested(alice, address(usdc)) == 7000
- The first investment of alice is not tracked via ftACL contract, leading to corrupted accounting

Impact

- Accounting Corruption - The amountInvested mapping becomes incorrect when users mix capped and unlimited proofs
- Per-User Cap Bypass - Users can exceed their per-user caps by using unlimited (proof amount = 0) proofs after capped proofs
- Inconsistent State - The ACL's view of user investments doesn't match reality

Proof of Concepts

Paste the following code here to run the test

```
import {ftACL} from "contracts/ftACL.sol";

function test_AccountingCorruption_MixedProofs() public {

        // Setup: User has two proofs
        address alice = makeAddr("alice");
        usdc.mint(alice, 1_000_000);
        vm.prank(alice);
        usdc.approve(address(manager), type(uint256).max);
        MerkleHelper.ACLEntry memory aliceEntry1 = MerkleHelper.ACLEntry({
            who: alice,
            asset: address(usdc),
            amount: 0
       });

       MerkleHelper.ACLEntry memory aliceEntry2 = MerkleHelper.ACLEntry({
            who: alice,
            asset: address(usdc),
            amount: 10_000 //10,000 USDC
       });

       MerkleHelper.ACLEntry[] memory entries = new MerkleHelper.ACLEntry[](2);
       entries[0] = aliceEntry1;
       entries[1] = aliceEntry2;
       bytes32[] memory aliceProofWithAmount0 = MerkleHelper.generateProof(entries, 0);
       bytes32[] memory aliceProofWithAmount10000 = MerkleHelper.generateProof(entries, 1);

       bytes32 merkleRoot = MerkleHelper.generateRoot(entries);
       ftACL acl = new ftACL(merkleRoot, address(manager));

```

```
31          vm.prank(msig);
32          manager.setACL(address(acl));
33
34           // Step 1: Invest with capped proof with amount 0
35           vm.prank(alice);
36           manager.invest(address(usdc), 5000, 0, aliceProofWithAmount0);
37           assertEq(acl.amountInvested(alice, address(usdc)), 0);
38
39           // Step 2: Invest with capped proof with amount 10000
40           vm.prank(alice);
41           manager.invest(address(usdc), 7000, 10000,
                aliceProofWithAmount10000);
42           assertEq(acl.amountInvested(alice, address(usdc)), 7000);
43
44           // wrapper total supply should be 12000, 5000 + 7000
45           // since alice invested 5000 with amount 0, it is not accounted
                 for amountInvested
46           // and alice invested again 7000 with amount 10000, it is
                accounted for amountInvested
47           assertEq(wrapper.totalSupply(), 12000);
48      }
```

Recommended mitigation

1. Prevent mixing proof types - If a user has invested with one proof type, prevent them from using a different proof type for the same token. This could be enforced by checking if amountInvested[user][token] > 0 and requiring the same proofAmount.
2. Separate tracking for unlimited proofs - Maintain a separate mapping for unlimited proof investments to keep accounting accurate.

[M-3] Malicious actors / MEV bots can monopolize the global cap of investment leading to DOS

Description

The `_invest` function in `PutManager.sol` checks the global collateralCap[token] at line 378, but this check is vulnerable to front-running attacks:

```
uint256 projected = collateralSupply[token] + amount;
if (collateralCap[token] != 0 && projected > collateralCap[token]) {
    revert ftPutManagerCollateralCapExceeded();
}
```

If there is a cap on the collateral, then anyone with investment close to cap can end up having their transaction submitted first by front running other transactions.

This is particularly problematic during public offerings where fair distribution is critical. A single user or MEV bot could consume the entire offering capacity before other whitelisted users can participate.

Root Cause:

The `_invest` function in `PutManager.sol` allows users with ACL proofs (`where proofAmount = 0`) can front-run other users' transactions (`where proofAmount > 0`). The per-user cap tracking is bypassed at line 393-396, allowing unlimited investments per user

```
if (address(ftACL) != address(0) && proofAmount != 0) {
    ftACL.invest(msg.sender, token, amount, proofAmount);
}
```

The invest function of ftACL would prevent user from investing more than proof-amount, but anyone with 0 proof amount can bypass this condition to get their investment amount checked.

```
 function invest(
        address account,
        address token,
        uint256 amount,
        uint256 proofAmount
    )
        external
        onlyPutManager
    {
        uint256 newInvestedAmount = amountInvested[account][token] +
            amount;
        if (newInvestedAmount > proofAmount) {
            revert ftACLCapReached();
        }

        amountInvested[account][token] = newInvestedAmount;
    }
```

Internal Pre-conditions:

- Configurator calls setCollateralCaps to set the collateral cap for a token (e.g. usdc)
- Global cap is set to 1,000,000 USDC for collateral USDC

Exterbnal Pre-conditions:

- Gas fee of User A's transaction needs to be very high to front run B's or any subsequent investments

Attack Path:

- User A (whale) has a proof: `(alice, USDC, 0)` with `proofAmount = 0`
- User B has a capped proof: `(bob, USDC, 10000)` with `proofAmount = 10000`

- User B submits transaction to invest 10,000 USDC
- User A monitors mempool and front-runs with transaction to invest 999,000 USDC
- User A's transaction succeeds, consuming 999,000 USDC of the cap
- User B's transaction reverts with ftPutManagerCollateralCapExceeded
- User A has successfully monopolized the global cap

Impact

- Denial of Service - Legitimate users with capped proofs can be prevented from investing
- Unfair Distribution - Users with unlimited proofs can monopolize the global cap
- MEV Advantage - Users who can pay higher gas fees (MEV bots) have unfair advantage
- Bypass of Intended Fairness - Per-user caps are meant to ensure fair distribution, but unlimited proofs bypass this entirely

Proof of Concepts

Paste the following code here to run the POC:

```
function test_front_Run_invest_by_users_with_proof_amount_0() external {

        //1. create proofs for 2 people one with proof amount alice - 0, bob - 10,000
        address alice = makeAddr("alice");
        usdc.mint(alice, 999_000);
        address bob = makeAddr("bob");
        usdc.mint(bob, 10_000);
        MerkleHelper.ACLEntry memory aliceEntry = MerkleHelper.ACLEntry({
            who: alice,
            asset: address(usdc),
            amount: 0 //0 USDC
        });

        MerkleHelper.ACLEntry memory bobEntry = MerkleHelper.ACLEntry({
            who: bob,
            asset: address(usdc),
            amount: 10_000 //10,000 USDC
        });

        MerkleHelper.ACLEntry[] memory entries = new MerkleHelper.ACLEntry[](2);
        entries[0] = aliceEntry;
        entries[1] = bobEntry;
        bytes32[] memory aliceProof = MerkleHelper.generateProof(entries, 0);
        bytes32[] memory bobProof = MerkleHelper.generateProof(entries, 1);
```

```
25
26          //Check the cap of usdc as global cap
27          //configurator updates the collateral cap for usdc to 1
                _000_000e6
28          vm.prank(configurator);
29          manager.setCollateralCaps(address(usdc), 1000000);
30          uint256 usdcCap = manager.collateralCap(address(usdc));
31          assertEq(usdcCap, 1000000); //Enabling unlimited investment
32
33          //bob's trying to make an investment
34          // vm.startPrank(bob);
35          // usdc.approve(address(manager), type(uint256).max);
36          // uint256 idBob = manager.invest({
37          //     token: address(usdc),
38          //     amount: 10_000e6,
39          //     proofAmount: 10_000e6,
40          //     proofWL: bobProof
41          // });
42          // vm.stopPrank();
43          // assertEq(pft.ownerOf(idBob), bob);
44
45          //show alice front running the transaction of bob by depositing
                a large amount of collateral
46          vm.startPrank(alice);
47          usdc.approve(address(manager), type(uint256).max);
48          uint256 idAlice = manager.invest({
49              token: address(usdc),
50              amount: 999_000,
51              proofAmount: 0,
52              proofWL: aliceProof
53          });
54          vm.stopPrank();
55          assertEq(pft.ownerOf(idAlice), alice);
56
57          vm.startPrank(bob);
58          usdc.approve(address(manager), type(uint256).max);
59          vm.expectRevert(PutManager.ftPutManagerCollateralCapExceeded.
                selector);
60          manager.invest({
61              token: address(usdc),
62              amount: 10_000e6,
63              proofAmount: 10_000e6,
64              proofWL: bobProof
65          });
66          vm.stopPrank();
67      }
```

Recommended mitigation

1. Enforce per-user caps even for unlimited proofs (proofAmount = 0)

- Track amountInvested for all users, including those with proofAmount = 0. Consider setting a reasonable default cap or requiring explicit cap values.

2. Implement commit-reveal scheme

- Use a two-phase investment process where users commit to their investment amount first, then reveal after a delay. This prevents front-running.

3. Add per-user minimum allocation

- Reserve a minimum allocation for each whitelisted user to ensure fair distribution. Reserve a minimum allocation for each whitelisted user to ensure fair distribution.

Low

[L-1] Silent truncation when casting to uint64 can cause acceptance of msig update sooner than intended

Description

When setMsig function of `PutManager.sol` is called, the `effectiveTime` is calculated as `block.timestamp + DELAY_MULTISIG`.

If the `effectiveTime` exceeds the maximum value of `uint64`, the value is truncated to the lower 64 bits. This can cause the `delayMsig` to be set to a value that is smaller than `block.timestamp`, allowing the msig update to be accepted sooner than intended.

```
1  function setMsig(address _msig) external onlyMsig {
2         if (_msig == address(0x0)) revert ftPutManagerZeroAddress();
3         if (_msig == msig || _msig == nextMsig) revert
             ftPutManagerInvalidMsig();
4         uint256 effectiveTime = block.timestamp + DELAY_MULTISIG;
5         emit MsigUpdateScheduled(_msig, effectiveTime);
6         nextMsig = _msig;
7         delayMsig = uint64(effectiveTime);
8     }
```

Concrete Example with Real Numbers

1. block.timestamp = 18,446,744,073,709,550,615 (1000 seconds before uint64.max)
2. DELAY_MULTISIG = 3600 seconds (1 hour)
3. Calculate effectiveTime

    - uint256 effectiveTime = block.timestamp + DELAY_MULTISIG;

4. Cast to uint64 (TRUNCATION OCCURS)
    - delayMsig = uint64(effectiveTime);
5. `acceptMsig()` Check Fails (Wrongly Allows Acceptance) when newMsig calls `acceptMsig()`:

```
if (msg.sender != nextMsig || block.timestamp < delayMsig) {
    revert ftPutManagerInvalidMsig();
}
```

- Since `block.timestamp < delayMsig` is FALSE, the condition `(msg.sender != nextMsig || block.timestamp < delayMsig)is FALSE`, so it doesn't revert.
- Result: `acceptMsig()` succeeds immediately, allowing the new admin to become admin in seconds instead of waiting for 1 hour.

Impact

This can allow an attacker to become the msig sooner than intended, potentially allowing them to take control of the contract.

- Expected: delay of 3,600 seconds (1 hour)
- Actual: delay of 0 seconds (immediate) due to truncation
- The 1-hour security delay is bypassed

When This Happens:

This requires `block.timestamp` to be extremely close to `uint64.max` (year ~584 billion), so it's not a practical concern in the near term. However, it's an unhandled edge case that should be fixed to prevent silent truncation.

Proof of Concepts

```
function testDelayMsigTruncationIssue() public {
        // Set block.timestamp to be very close to uint64 max
        // When we add DELAY_MULTISIG, it will exceed uint64 max and truncate
        Fixture memory fix = _deployFixture();
        uint256 DELAY_MULTISIG = 1 hours;

        uint64 UINT64_MAX = type(uint64).max;
        uint256 blockTimestamp = UINT64_MAX - 1000; // 1000 seconds before uint64 max
        vm.warp(blockTimestamp);

        vm.startPrank(fix.msig);
        fix.manager.setMsig(fix.newMsig);
        vm.stopPrank();
```

```
14
15          uint64 effectiveTime = uint64(blockTimestamp + DELAY_MULTISIG);
16
17          console2.log("Actual delay in seconds:", effectiveTime);
18          console2.log("Expected delay (1 hour):", DELAY_MULTISIG);
19          console2.log("Delay difference:", effectiveTime <
                DELAY_MULTISIG ? "SHORTER" : "LONGER");
20
21          // In this case, the delay is much shorter than 1 hour due to
                truncation
22          if (effectiveTime < DELAY_MULTISIG) {
23              console2.log(" SECURITY ISSUE: Delay is shorter than
                    intended!");
24          }
25      }
```

Logs:

- Actual delay in seconds: 2599
- Expected delay (1 hour): 3600
- Delay difference: SHORTER
- SECURITY ISSUE: Delay is shorter than intended!

Recommended mitigation

- Use a larger data type for the `delayMsig` variable.
- Add a check to ensure that the `effectiveTime` is not greater than the maximum value of `uint64`.

```
1
2   function setMsig(address _msig) external onlyMsig {
3       if (_msig == address(0x0)) revert ftPutManagerZeroAddress();
4       if (_msig == msig || _msig == nextMsig) revert
            ftPutManagerInvalidMsig();
5       uint256 effectiveTime = block.timestamp + DELAY_MULTISIG;
6
7       // Add validation to prevent truncation
8       if (effectiveTime > type(uint64).max) revert
            ftPutManagerInvalidMsig();
9
10      emit MsigUpdateScheduled(_msig, effectiveTime);
11      nextMsig = _msig;
12      delayMsig = uint64(effectiveTime);
13  }
```

[L-2] CircuitBreaker main buffer cap uses stale TVL, creating design trade-off between flash loan protection and rate limiting accuracy

Description

- The CircuitBreaker contract calculates the main buffer cap using preTvl (TVL before the operation) instead of the post-operation TVL.
- This is an intentional design choice to provide flash loan attack protection (as demonstrated in `test_FlashloanDoS_MainBufferUnaffected`), but it creates a design limitation where the main buffer cap becomes stale for legitimate operations, leading to slightly inaccurate rate limiting over time.

Design Intent:

The circuit breaker uses a dual-buffer system:

- **Main Buffer:** Time-replenishing buffer, capped by `TVL * maxDrawRateWad` (intended to represent baseline capacity)
- **Elastic Buffer:** Deposit-tracking buffer that increases with deposits and decays over time (temporary capacity)

Flash Loan Protection (Current Design):

Using `preTvl` prevents flash loan attacks from immediately increasing the main buffer cap:

- Flash loan deposit of 100k tokens → Main buffer cap stays at baseline TVL (protected)
- Only elastic buffer increases (temporary, decays over time)

NOTE:

For legitimate operations, the main buffer cap uses stale TVL until the next operation. If this is intended then it should be low else it should be considered a Medium severity issue for causing inconsistency between expected and actual rate limits.

ftYieldWrapper.sol

```
// ftYieldWrapper.sol
uint256 preTvl = valueOfCapital();  // TVL = 1,000,000
recordInflow(token, amount, preTvl);  // amount = 100,000
```

CircuitBreaker.sol

```
// Inside recordInflow:
_updateBuffers(asset, state, preTvl);  // Uses 1,000,000 for cap
// Cap = (1,000,000 * maxDrawRateWad) / 1e18 = 100,000

```

```
// But actual TVL after deposit = 1,100,000
// Cap remains at 100,000 until next operation
```

Impact

- *Stale Cap Calculations:* The main buffer cap does not immediately reflect the current TVL after operations complete, creating a lag between actual TVL and cap calculation
- *Slightly Inaccurate Rate Limiting:* After legitimate deposits, the cap is temporarily lower than it could be (underestimates capacity) After withdrawals, the cap is temporarily higher than it could be (overestimates capacity)
- *Accumulating Minor Error:* Over multiple operations, the cap calculation can become slightly inaccurate relative to actual TVL
- *Design Trade-off:* This is an intentional trade-off - flash loan protection is prioritized over instant accuracy

Proof of Concepts

```
    function test_TVL_Inconsistency_CapCalculation() public {
        // Setup: 10% max draw rate, 1 hour main window
        cb = new CircuitBreaker(0.1e18, 1 hours, 2 hours);
        cb.addProtectedContract(wrapper);

        uint256 initialTvl = 1_000_000e6; // 1M USDC
        uint256 depositAmount = 100_000e6; // 100K USDC
        uint256 withdrawalAmount = 50_000e6; // 50K USDC

        // Expected caps based on TVL
        uint256 expectedCapAfterDeposit = (initialTvl + depositAmount)
            * 0.1e18 / 1e18; // 110,000
        uint256 expectedCapAfterWithdrawal = (initialTvl +
            depositAmount - withdrawalAmount) * 0.1e18 / 1e18; //
            105,000

        // Step 1: Record inflow with preTvl = initialTvl
        // Current code uses preTvl for cap calculation
        vm.prank(wrapper);
        cb.recordInflow(address(asset), depositAmount, initialTvl);

        // Get the state after deposit
        (uint256 mainBuffer1,,) =
            cb.getRawAssetState(address(asset));

        // Check: The cap was calculated using preTvl (1,000,000), not
            postTvl (1,100,000)
        // If cap was calculated correctly, mainBuffer would be capped
            at 110,000
        // But since it used 1,000,000, the cap is 100,000
        // However, on first interaction, mainBuffer is set to cap, so
```

```
            it should be 100,000
        uint256 calculatedCap1 = initialTvl * 0.1e18 / 1e18; // 100,000 (WRONG - uses old TVL)
        assertEq(mainBuffer1, calculatedCap1, "Main buffer should equal cap calculated from preTvl");

        // The issue: mainBuffer1 = 100,000, but it should be 110,000
        // This demonstrates the cap is calculated using stale TVL
        assertLt(mainBuffer1, expectedCapAfterDeposit, "Cap is lower than it should be after deposit");

        // Step 2: Check and record outflow with preTvl = initialTvl + depositAmount
        // But the withdrawal will reduce TVL to initialTvl + depositAmount - withdrawalAmount
        uint256 preTvlBeforeWithdrawal = initialTvl + depositAmount; // 1,100,000

        vm.prank(wrapper);
        (bool allowed,) = cb.checkAndRecordOutflow(
            address(asset),
            withdrawalAmount,
            preTvlBeforeWithdrawal
        );

        assertTrue(allowed, "Withdrawal should be allowed");


        // Check: The cap was calculated using preTvl (1,100,000), not postTvl (1,050,000)
        // The cap calculation in _updateBuffers used preTvlBeforeWithdrawal
        // So cap = 1,100,000 * 0.1 = 110,000
        // But it should be = 1,050,000 * 0.1 = 105,000
        uint256 calculatedCap2 = preTvlBeforeWithdrawal * 0.1e18 / 1e18; // 110,000 (WRONG - uses old TVL)

        // The main buffer was replenished based on the wrong cap
        // After withdrawal of 50,000 from elastic buffer (which had 100,000),
        // elastic buffer = 50,000, main buffer unchanged at 100,000
        // Then _updateBuffers calculates replenishment based on cap = 110,000
        // But the actual TVL after withdrawal is 1,050,000, so cap should be 105,000

        // This demonstrates the cap is calculated using stale TVL (pre-withdrawal instead of post-withdrawal)
        assertGt(calculatedCap2, expectedCapAfterWithdrawal, "Cap is higher than it should be after withdrawal");

```

```
        // Step 3: Verify the issue persists over multiple operations
        // After the operations:
        // - Actual TVL = 1,050,000
        // - Cap should be = 105,000
        // - But cap was calculated as 110,000 (using pre-withdrawal TVL)

        // Query withdrawal capacity using the actual current TVL
        uint256 actualCurrentTvl = initialTvl + depositAmount - withdrawalAmount; // 1,050,000
        uint256 capacity = cb.withdrawalCapacity(address(asset), actualCurrentTvl);

        // The capacity calculation uses _calculateBuffers which recalculates based on currentTvl
        // So this should be correct, but the stored state (mainBuffer, elasticBuffer)
        // was updated using the wrong TVL

        // The stored mainBuffer was updated based on cap = 110,000 (wrong)
        // But when querying capacity with actualCurrentTvl = 1,050,000,
        // _calculateBuffers recalculates cap = 105,000 (correct)
        // This creates inconsistency between stored state and view calculations

        console2.log("Initial TVL:", initialTvl);
        console2.log("After deposit - Expected cap:", expectedCapAfterDeposit);
        console2.log("After deposit - Actual cap (from preTvl):", calculatedCap1);
        console2.log("After withdrawal - Expected cap:", expectedCapAfterWithdrawal);
        console2.log("After withdrawal - Actual cap (from preTvl):", calculatedCap2);
        console2.log("Current TVL:", actualCurrentTvl);
        console2.log("Capacity query (uses correct TVL):", capacity);
    }
```

Logs:

```
Initial TVL: 1000000000000
After deposit - Expected cap: 110000000000
After deposit - Actual cap (from preTvl): 100000000000
After withdrawal - Expected cap: 105000000000
After withdrawal - Actual cap (from preTvl): 110000000000
Current TVL: 1050000000000
Capacity query (uses correct TVL): 150000000000
```

Recommended mitigation

If accuracy is desired, consider implementing a gradual TVL update mechanism that maintains flash loan protection while improving accuracy:

- Track a "baseline TVL" that updates gradually (e.g., using exponential moving average or time-weighted average)
- Use this baseline TVL for main buffer cap calculation instead of preTvl
- This maintains flash loan protection (baseline doesn't instantly increase) while gradually updating the cap

[L-3] CircuitBreaker rate limit based on total value (including yield) creates mismatch with actual withdrawable capacity

Description

The `CircuitBreaker` calculates `withdrawal rate limits` using `valueOfCapital()` (capital + yield) as TVL, but users can only actually withdraw `availableToWithdraw()` (principal only, capped at `totalSupply()`). This creates a design question: should the rate limit reflect total value or actual withdrawable capacity?

The Mismatch

What Circuit Breaker Uses: ftYieldWrapper::deposit

```
1  // ftYieldWrapper
2    function deposit(uint256 amount) external nonReentrant
        onlyPutManagerOrDepositor {
3          if (amount == 0) revert ftYieldWrapperInsufficientLiquidity();
4
5          // Circuit breaker: record inflow (fail-open)
6          address _cb = circuitBreaker;
7          if (_cb != address(0)) {
8  @>      uint256 preTvl = valueOfCapital(); //Capital + yield
9              try ICircuitBreaker(_cb).recordInflow(token, amount, preTvl
                  ) {} catch {}
10         }
11
12         IERC20(token).safeTransferFrom(msg.sender, address(this),
              amount);
13         _mint(msg.sender, amount);
14         emit Deposit(msg.sender, amount);
15     }
```

What's Actually Withdrawable:

ftYieldWrapper::availableToWithdraw

```
function availableToWithdraw(address strategy) public view returns (uint256 liquidity) {
        // Only consider registered strategies; unrecognized addresses return zero
        if (!isStrategy(strategy)) return 0;
        // Defensive: treat failing strategies as having zero liquidity
        uint256 shares;
        try IStrategy(strategy).balanceOf(address(this)) returns (uint256 sb) {
            shares = sb;
        } catch {
            return 0;
        }
@>       try IStrategy(strategy).maxAbleToWithdraw(shares) returns (uint256 m) {
            liquidity = m;
        } catch {
            return 0;
        }
    }
```

Example Scenario:

Initial: principal = 1,000, yield = 0

- valueOfCapital() = 1,000
- availableToWithdraw() = 1,000
- Buffer cap = 1,000 × 5% = 50 (matches)

After yield accrues: principal = 1,000, yield = 100

- valueOfCapital() = 1,100 (includes yield)
- availableToWithdraw() = 1,000 (capped at principal)
- Buffer cap = 1,100 × 5% = 55 (based on total value)
- Actual withdrawable capacity = 1,000 × 5% = 50 (based on principal)
- Mismatch: Buffer suggests 55, but only 50 is actually withdrawable

Design Question

This could be intentional as a **two-layer protection system**: 1. **Circuit Breaker (Rate Limiter):** Limits withdrawal rate based on total value 2. **Wrapper (Liquidity Checker):** Ensures actual funds are available

However, this creates a disconnect where:

- The rate limit suggests more capacity than is actually withdrawable
- As yield accrues, the gap widens (rate limit becomes less meaningful)
- Users/admins might misinterpret buffer capacity as actual withdrawable amount

Impact

- Rate Limit Becomes Less Meaningful: As yield accrues, the rate limit is based on total value but actual withdrawals are capped at principal, making the limit less effective for real withdrawals
- Potential Confusion: Buffer capacity suggests more than is actually withdrawable, which could confuse users or admins monitoring the system
- Design Clarity: The mismatch between rate limit (total value) and actual capacity (principal) isn't clearly documented

Recommended mitigation If rate limit should reflect actual capacity then modify to use `availableToWithdraw()` for TVL:

```
// In ftYieldWrapper.sol
 - uint256 preTvl = valueOfCapital();
 + uint256 preTvl = availableToWithdraw();  // Use actual withdrawable amount
cb.recordInflow(token, amount, preTvl);
cb.checkAndRecordOutflow(token, amount, preTvl);
```

This would make the rate limit more meaningful for actual withdrawals, but note that `availableToWithdraw()` can change between the circuit breaker check and the actual withdrawal.

Informational

[I-1] `yieldWrapper::withdrawQueued()` and `yieldWrapper::claimQueued()` will always revert when called with strategies that don't implement `IStrategyWithQueue`