Collar Protocol Security Review

Auditors

MiloTruck, Lead Security Researcher

R0bert, Lead Security Researcher

Om Parikh, Security Researcher

0xDjango, Associate Security Researcher

Report prepared by: Lucas Goiriz

January 15, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

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.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Collar Protocol according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- 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.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

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

4 Executive Summary

Over the course of 14 days in total, Collar Protocol engaged with Spearbit to review the protocol-core protocol. In this period of time a total of **13** issues were found.

Summary

Project Name	Collar Protocol
Repository	protocol-core
Commit	d8ee1d2d
Type of Project	DeFi, Lending
Audit Timeline	Dec 27th to Jan 10th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	1	0
Low Risk	5	3	2
Gas Optimizations	0	0	0
Informational	7	4	3
Total	**13**	**8**	**5**

5 Findings

5.1 Medium Risk

5.1.1 Borrowers can force swaps on escrow suppliers by transferring the loan's NFT to them

Severity: Medium Risk

Context: LoansNFT.sol#L264-L265, LoansNFT.sol#L425-L426

Description: In `LoansNFT.closeLoan()`, `_isSenderOrKeeperFor()` is called to check if the borrower has approved the keeper to close the loan on their behalf:

```
address borrower = ownerOf(loanId);
require(_isSenderOrKeeperFor(borrower, loanId), "loans: not NFT owner or allowed keeper");
```

Similarly, in `LoansNFT.forecloseLoan()`, `_isSenderOrKeeperFor()` is called to check if the escrow supplier allows the keeper to call `forecloseLoan()` on their behalf:

```
address escrowOwner = escrowNFT.ownerOf(escrowId);
require(_isSenderOrKeeperFor(escrowOwner, loanId), "loans: not escrow owner or allowed keeper");
```

However, since both `closeLoan()` and `forecloseLoan()` use the same `keeperApprovedFor` mapping to check for keeper approval, borrowers can forcefully swap `cashAsset` held by the escrow supplier into `underlying` without their permission.

This is achieved by directly transferring the loan NFT to the escrow supplier, for example:

- Assume Bob has a dangling `cashAsset` allowance to the `LoansNFT` contract.
- Alice calls `openLoan()` to open an escrowed loan with Bob as the escrow supplier. The loan has a `loanId` of `1337`.
- Bob calls `setKeeperApproved()` for `loanId = 1337` to allow the keeper to call `forecloseLoan()` on his behalf.
- Alice transfers her loan NFT to Bob.
- After loan expiry, the keeper automatically calls `closeLoan()` to close the loan on Bob's behalf:
  - The `_isSenderOrKeeperFor(borrower, loanId)` check passes as `keeperApprovedFor[Bob][1337] = true`.
  - `loanAmount` of `cashAsset` is pulled from Bob and swapped to `underlyingAsset` without his permission.

Note that for this exploit to occur, the following conditions must be met:

1. The escrow supplier has a dangling approval of `cashAsset` to the `LoansNFT` contract.
2. The loan must be escrowed, so the attacker pays interest for the loan.
3. The escrow supplier has to actively call `setKeeperApproved()` after the loan is created.

However, this scenario is still entirely plausible if the loan was created without malicious intention but happens to become non-profitable for the borrower after expiration.

For example, in the scenario above, Alice opens the escrowed loan believing that the price of `underlyingAsset` will increase and has no intentions to perform such an attack. However, after expiration, she sees that calling `closeLoan()` is non-profitable or would in a loss for her (eg. `underlyingAsset` price goes below `putStrikePrice` or the loan incurs late fees), so she performs this attack to grief the escrow supplier.

Recommendation: Consider having two mappings (eg. `keeperApprovedForBorrower` and `keeperApprovedForEscrowSupplier`) to separate borrower and escrow supplier approvals. `keeperApprovedForBorrower` would be used in `closeLoan()` while `keeperApprovedForEscrowSupplier` is used in `forecloseLoan()`.

Collar: Fixed in commit 897a702d by removing foreclosure flow and complexity entirely.

The likelihood is very low, since even for the few users that end up in this situation there is a single target they can grief, and that escrow owner needs to also have a sufficient cash approval and balance to Loans (even though they are a supplier of underlying), and have approved a keeper for foreclosing that specific loan. The attacker has 1 day when this is possible, since if their position is empty, foreclosure (by keeper) will happen after min grace.

Spearbit: Verified, `forecloseLoan()` has been removed in the new protocol design, as such, this issue is resolved as escrow suppliers no longer grant approval to the keeper.

5.2 Low Risk

5.2.1 Missing validation checks in `LoansNFT._executeRoll()`

Severity: Low Risk

Context: LoansNFT.sol#L736-L740

Description: In `LoansNFT._executeRoll()`, only `canOpenPair()` is checked for the `rolls` contract, as shown below:

```
(Rolls rolls, uint rollId) = (rollOffer.rolls, rollOffer.id);
// check this rolls contract is allowed
require(configHub.canOpenPair(underlying, cashAsset, address(rolls)), "loans: unsupported rolls");
// taker matching roll's taker is not checked because if doesn't match, roll should check / fail
// offer status (active) is not checked, also since rolls should check / fail
```

The function currently does not explicitly check the following conditions, and instead relies on `rolls.executeRoll()` to revert:

- `takerNFT` in both contracts match (i.e. `rolls.takerNFT() == takerNFT`).
- `cashAsset` in both contracts match (i.e. `rolls.cashAsset() == cashAsset`).
- The `takerId` stored in `rollId` matches `loanId` (i.e. `rolls.getRollOffer(rollId).takerId == _takerId(loanId)`).

If `takerNFT`, `cashAsset` or `takerId` do not match, `executeRoll()` reverts as there is no approval from `LoansNFT` to `Rolls` for `rolls.takerNFT`. However, this is not a strong guarantee.

Recommendation: Explicitly check the conditions listed above:

```
  // check this rolls contract is allowed
  require(configHub.canOpenPair(underlying, cashAsset, address(rolls)), "loans: unsupported rolls");
- // taker matching roll's taker is not checked because if doesn't match, roll should check / fail
- // offer status (active) is not checked, also since rolls should check / fail
+ require(rolls.takerNFT() == takerNFT, "loans: takerNFT mismatch");
+ require(rolls.cashAsset() == cashAsset, "loans: cashAsset mismatch");
+ require(rolls.getRollOffer(rollId).takerId == _takerId(loanId), "loans: takerId mismatch");
```

Collar: Fixed in commit 897a702d by adding some of the suggested checks (and some additional ones).

Spearbit: Verified, the `takerNFT` check has been added, which implicitly guarantees `cashAsset` in both contracts also match. Note that the roll offer's `takerId` is still not checked to match `loanId`.

5.2.2 Provider has no control over protocol fees paid when collar positions are created

Severity: Low Risk

Context: CollarProviderNFT.sol#L242-L243

Description: When `CollarProviderNFT.mintFromOffer()` is called to create a collar position, the protocol fee paid by the provider depends on what `ConfigHub.protocolFeeAPR` is set to whenever the function is called:

```
// calc protocol fee to subtract from offer (on top of amount)
(uint fee, address feeRecipient) = protocolFee(providerLocked, offer.duration);
```

As such, the provider has no control over the protocol fee he pays, for example:

- `protocolFeeAPR` is 0.1%.
- A provider creates an offer, thinking that the current 0.1% fee is acceptable.
- Protocol owner increases `protocolFeeAPR` to 1%.
- When `mintFromOffer()` is called, the provider pays a 1% fee instead of 0.1%.

This could be misleading as providers could assume they will pay the current `protocolFeeAPR` when creating offers. However, note that this risk is fairly mitigated as `protocolFeeAPR` can only be set to a maximum of 1%.

Recommendation: Consider documenting this behavior so that providers are aware - whenever they create an offer, they should be willing to pay up to a 1% protocol fee.

Collar: Fixed in commit 897a702d by adding documentation regarding this behavior.

Provider offers are assumed to be actively managed w.r.t market conditions - mainly asset price changes, but also other factors impacting offer terms, with protocol fee being one such factor. Additionally, it is assumed to be changed infrequently and adequate prior warning, and the range in ConfigHub is limited to below 1%.

Spearbit: Verified.

5.2.3 Centralization risk due to `BaseManaged.rescueTokens()`

Severity: Low Risk

Context: BaseManaged.sol#L74-L85

Description: All contracts in the protocol inherit `BaseManaged`, which contains the `rescueTokens()` function:

```
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);
}
```

However, this function allows the owner to transfer out all funds in the protocol, including any loan/provider/taker/escrow NFTs held in the contracts.

Recommendation: Consider checking that `token` is not `cashAsset`/`underlying`/`address(this)` in their respective contracts. This prevents the owner from transferring user funds out of the protocol.

Collar: Acknowledged. This is unfortunately true, but the intention is to provide safety against a catastrophic bug, in which case cash or underlying is what needs to be rescued. The centralisation risk in this case is similar to an upgradeable contract, but with surface area (operational risks, and smart contract risk) reduced.

We've added more explicit documentation around these concerns in commit 897a702d

Spearbit: Acknowledged.

5.2.4 `ConfigHub.canOpenPair()` does not guarantee contracts are of the correct type

Severity: Low Risk

Context: CollarTakerNFT.sol#L172-L177, CollarProviderNFT.sol#L233-L234, LoansNFT.sol#L613-L617, LoansNFT.sol#L737-L738

Description: Throughout the protocol, `ConfigHub.canOpenPair()` is called to check that addresses passed by the user are valid and whitelisted by the protocol. For example, `CollarTakerNFT.openPairedPosition()` calls `canOpenPair()` to check that the specified `providerNFT` address is whitelisted:

```
// check asset & self allowed
require(configHub.canOpenPair(underlying, cashAsset, address(this)), "taker: unsupported taker");
// check assets & provider allowed
require(
    configHub.canOpenPair(underlying, cashAsset, address(providerNFT)), "taker: unsupported provider"
);
```

However, since the `canOpenSets[underlying][cashAsset]` set contains the `CollarProviderNFT`, `CollarTakerNFT`, `LoanNFT` and `Rolls` contract addresses, these `canOpenPair()` checks are not safe.

Using the check above as an example, the call to `canOpenPair()` only validates that the `providerNFT` address is whitelisted. However, there is nothing checking that `providerNFT` is actually the address `CollarProviderNFT` address. An attacker could pass the corresponding `CollarTakerNFT` address (or any of the other contract addresses) and this check would still pass.

This could be problematic if the intended contract and the contract passed have functions with the same signature or a function selector collision (eg. `CollarTakerNFT` is passed here instead of `CollarProviderNFT` and both contracts have a `getOffer()` function), since external calls to the specified address will not revert.

Note that in the current implementation of the codebase, all contracts do not have functions with the same signature (excluding functions from `BaseNFT`/`BaseManaged`) or colliding function selectors.

Recommendation: Consider implementing separate mappings for each type of contract, instead of storing all types of contracts in a single `canOpenSets` mapping. Alternatively, add an explicit "type" view check to each contract, similar to how ERC-165 is implemented.

Collar: Acknowledged. We think the safety in practice is sufficient since even if selectors would collide, effects of mutative methods need to also match. In the case of `providerNFT`, its `mintFromOffer` needs to return a decodable uint (as `providerId`), and later `expiration()` view needs to return the right expiration for that ID. In other contracts as well, effects of mutative methods prevent a confusion between the different types of internal contracts.

We'll keep this to balance specificity with simplicity, but have added some additional validations in commit 897a702d.

Spearbit: Acknowledged.

5.2.5 Rolling loans with price outside the put-call range breaks the `LTV == putStrikePercent` assumption

Severity: Low Risk

Context: Rolls.sol#L445, LoansNFT.sol#L968-L977

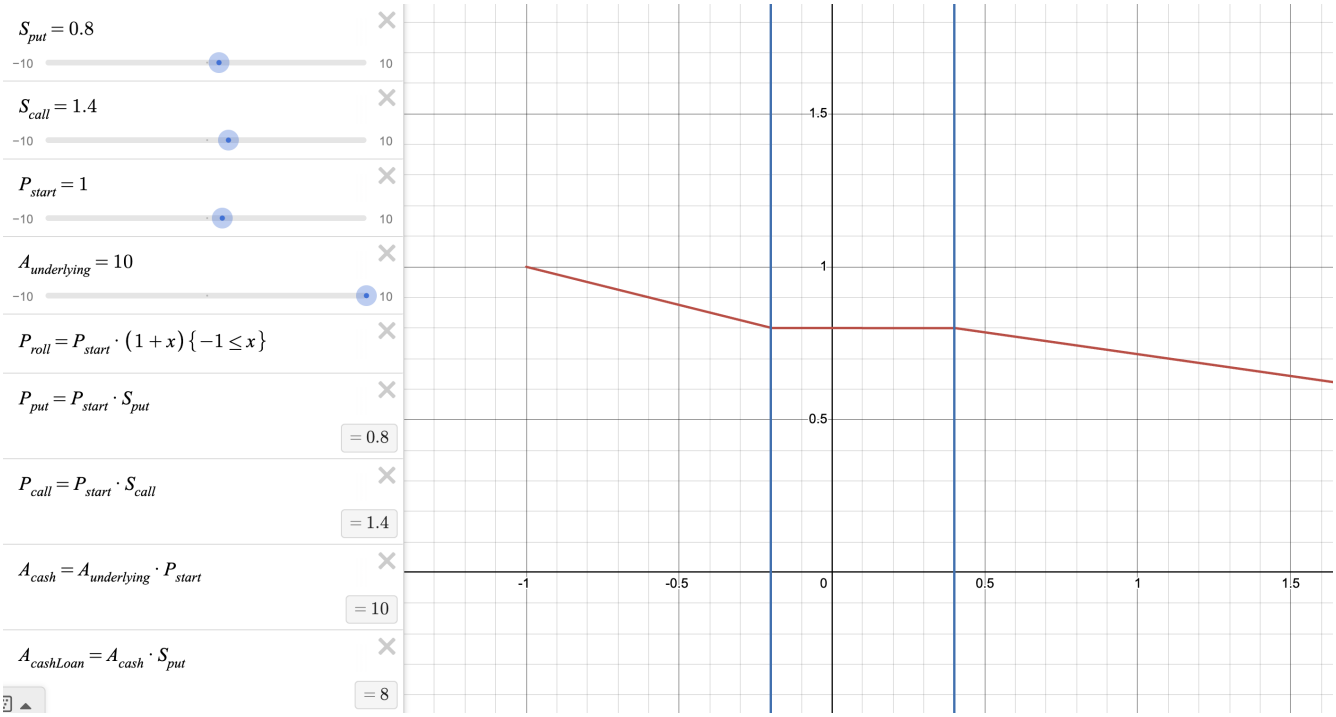
Description: When loans are rolled, the new `takerLocked` amount is calculated based on the change in price:

```
newTakerLocked = takerPos.takerLocked * newPrice / takerPos.startPrice;
```

Afterwards, `takerSettled - newTakerLocked` (equal to `fromRollsToUser + rollFee` below, where `takerSettled` is the amount the taker receives when the previous collar position is settled) is added to `loanAmount` for the new loan:

```
int loanChange = fromRollsToUser + rollFee;
if (loanChange < 0) {
    uint repayment = uint(-loanChange); // will revert for type(int).min
    require(repayment <= prevLoanAmount, "loans: repayment larger than loan");
    // if the borrower manipulated (sandwiched) their open swap price to be very low, they
    // may not be able to roll now. Rolling is optional, so this is not a problem.
    newLoanAmount = prevLoanAmount - repayment;
} else {
    newLoanAmount = prevLoanAmount + uint(loanChange);
}
```

However, this calculation breaks the assumption that a loan's LTV (i.e. `loanAmount` proportional to `takerLocked`) is always equal to `putStrikePercent`. With reference to the following graph:



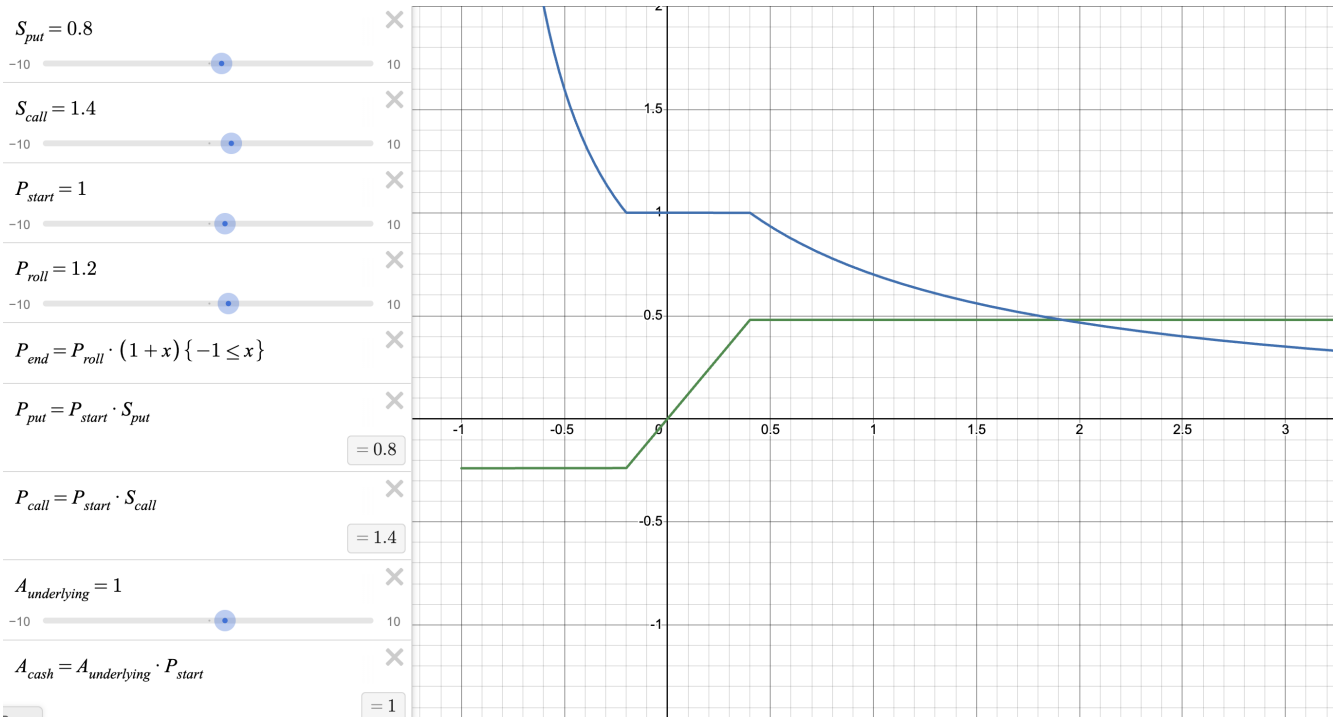
where:

- x is the % change in price (eg. `x = 0.1` is a 10% increase in price).
- y is the LTV after the loan is rolled (ie. red line).
- Blue lines are `putStrikePercent` and `callStrikePercent`.

As seen in the graph, the `LTV == putStrikePercent` assumption breaks when the loan is rolled while price is outside the `[putStrikePrice, callStrikePrice]` range. When this occurs, it has the following impacts on the new loan:

1. The new loan's LTV could be outside of LTV bounds (i.e. `[minLTV, maxLTV]`) specified in `ConfigHub`.
2. If price keeps increasing and a loan is repeatedly rolled while the price is above `callStrikePrice`, the check in `_loanAmountAfterRoll()` (LoansNFT.sol#L971) would eventually revert due to the previous `loanAmount` becoming too small.
3. Loans no longer maintain 100% exposure to `underlyingAsset` when LTV is not equal to `putStrikePercent`.

With reference to the following graph:



which simulates the loan being rolled and closed afterwards, where:

- `P_roll` is the price at which the loan is rolled.
- x is the % change in price from `P_roll` (ie. after the loan is rolled).
- Green line is the taker's profit/loss from the collar position in `cashAsset` when the loan is closed after it is rolled once.
- Blue line the is amount of `underlying` the taker receives upon closing the loan.

When `P_roll` is within `[putStrikePrice, callStrikePrice]`, the amount of `underlying` the taker receives upon closing the loan is the same as the amount of `underlying` the loan was initially opened with. However, once `P_roll` is outside the put-call range (eg. `P_roll = 1.6`), the amount of `underlying` received is no longer equal to the initial amount as the blue line is no longer constant.

Recommendation: Disallow loans to be rolled when price is outside the `[putStrikePrice, callStrikePrice]` range. Alternatively, document that loans rolled outside the put-call range break the `LTV == putStrikePercent` assumption and has the behaviors listed above.

Collar: Fixed in commit 897a702d by documenting this behavior. Currently there is no adverse impact, while disallowing this would limit the usefulness of the protocol for trending markets.

Spearbit: Verified.

5.3 Informational

5.3.1 Minor improvements to code and comments

Severity: Informational

Context: BaseManaged.sol#L30-L31, LoansNFT.sol#L74-L75, LoansNFT.sol#L183, LoansNFT.sol#L530-L531, LoansNFT.sol#L685-L687, CollarTakerNFT.sol#L196-L205, BaseNFT.sol#L39, ConfigHub.sol#L41, CollarProviderNFT.sol#L326, CollarProviderNFT.sol#L196-L206, EscrowSupplierNFT.sol#L254-L265

Description/Recommendation:

1. BaseManaged.sol#L30-L31: It is best practice to initialize all of a contract's state variables through its constructor. Consider passing `_newConfigHub` into the constructor here and calling `_setConfigHub()`, otherwise, future contracts inheriting `BaseManaged` might forget to call `_setConfigHub()`.
2. LoansNFT.sol#L74-L75: The `defaultSwapper` address is never used in any of the contract's functionality. Consider removing it and be storing it off-chain instead.
3. LoansNFT.sol#L183: `noEscrow` does not need to be initialized here as it already contains `address(0)` and `0` by default:

```
- EscrowOffer memory noEscrow = EscrowOffer(EscrowSupplierNFT(address(0)), 0);
+ EscrowOffer memory noEscrow;
```

4. LoansNFT.sol#L530-L531: The code here can be simplified:

```
if (allow) {
    require(bytes(ISwapper(swapper).VERSION()).length > 0, "loans: invalid swapper");
    allowedSwappers.add(swapper);
} else {
    allowedSwappers.remove(swapper);
}
```

5. LoansNFT.sol#L685-L687: The code here can be simplified since `amountOut` is initialized to `0` by default:

```
- if (amountIn == 0) {
-     amountOut = 0;
- } else {
+ if (amountIn != 0) {
```

6. CollarTakerNFT.sol#L196-L205: Consider incrementing and caching `nextTokenId` before using it in `mintFromOffer()`:

```
+ // increment ID
+ takerId = nextTokenId++;  // open the provider position for providerLocked amount (reverts if can't).

  // open the provider position for providerLocked amount (reverts if can't).
  // sends the provider NFT to the provider
- providerId = providerNFT.mintFromOffer(offerId, providerLocked, nextTokenId);
+ providerId = providerNFT.mintFromOffer(offerId, providerLocked, takerId);

  // check expiration matches expected
  uint expiration = block.timestamp + offer.duration;
  require(expiration == providerNFT.expiration(providerId), "taker: expiration mismatch");

- // increment ID
- takerId = nextTokenId++;
```

7. BaseNFT.sol#L39: `base` should be declared as a constant.
8. ConfigHub.sol#L41: Consider using `5 minutes` instead of `300` for readability:

```
- uint public constant MIN_CONFIGURABLE_DURATION = 300; // 5 minutes
+ uint public constant MIN_CONFIGURABLE_DURATION = 5 minutes;
```

9. CollarProviderNFT.sol#L326: Typo: "triggerred" → "triggered"
10. CollarProviderNFT.sol#L196-L206, EscrowSupplierNFT.sol#L254-L265: The code can be simplified by setting `offer.available` to `newAmount` outside the if-blocks, for example:

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

Collar: Implemented some of the suggestions in commit 897a702d.

Spearbit: Verified, some recommendations have been implemented where deemed appropriate.

5.3.2 `LoansNFT._conditionalEndEscrow()` could revert when performing a zero approval

Severity: Informational

Context: LoansNFT.sol#L851-L856

Description: When `LoansNFT._conditionalEndEscrow()` is called, it approves the `escrowNFT` contract to spend `toEscrow` amount of `underlying` tokens:

```
// if owing more than swapped, use all, otherwise just what's owed
uint toEscrow = Math.min(fromSwap, totalOwed);
// if owing less than swapped, left over gains are for the borrower
uint leftOver = fromSwap - toEscrow;

underlying.forceApprove(address(escrowNFT), toEscrow);
```

However, it is theoretically possible for `toEscrow` to be `0` if:

1. A loan with `takerLocked = 0` and `loanAmount = 0` is opened.
2. `closeLoan()` is called before expiration, so there are no late fees.

This would make `fromSwap`, `totalOwed` and `lateFee` all be `0`. When that happens, a call to `approve()` with zero value will be made, which reverts for tokens such as BNB on mainnet:

```
/* Allow another contract to spend some tokens in your behalf */
function approve(address _spender, uint256 _value)
    returns (bool success) {
if (_value <= 0) throw;
    allowance[msg.sender][_spender] = _value;
    return true;
}
```

As such, it might not be possible to close an escrowed loan with zero value.

Recommendation: If the protocol is deployed to mainnet in the future, consider making the following change:

```
- underlying.forceApprove(address(escrowNFT), toEscrow);
+ if (toEscrow != 0) underlying.forceApprove(address(escrowNFT), toEscrow);
```

Collar: Acknowledged. We intend to only integrate tokens that allow 0 approvals. Additionally, a 0 amount swap in closing is not a plausible scenario (unlikely even with 1 wei underlying), and has no impact (due to the dust amounts).

Spearbit: Acknowledged.

5.3.3 Additional safety checks and validation

Severity: Informational

Context: EscrowSupplierNFT.sol#L456-L461, CollarTakerNFT.sol#L238

Description/Recommendation:

1. EscrowSupplierNFT.sol#L456-L461: In `setLoansCanOpen()`, consider checking that the `underlying` token of the `loans` contract is the same as `asset` (i.e. `loans.underlying() == asset`). This prevents the owner from mistakenly whitelisting a `loans` contract with a different `underlying` token.
2. CollarTakerNFT.sol#L238: Consider checking `position.expiration != 0` as well for consistency. This prevents `block.timestamp >= expiration` from incorrectly passing when `expiration` is `0`:

```
- require(block.timestamp >= position.expiration, "taker: not expired");
+ require(position.expiration != 0 && block.timestamp >= position.expiration, "taker: not expired");
```

Collar: Acknowledged. We've decided against adding these:

- `setLoansCanOpen`: After the upfront late fee change, the escrow's dependence on loans functionality is even lower, and there isn't an impact to escrow from this (and loans checks on its side).
- `position.expiration != 0`: the check in provider NFT is needed to ensure position exists, however in taker (CollarTakerNFT.sol#L235-L236) this is ensured in `getPosition` (CollarTakerNFT.sol#L83), and so expiration cannot be 0 (unless that check is removed).

Spearbit: Acknowledged.

5.3.4 `SwapperUniV3.swap()` can be forced to leave a dangling approval to Uniswap's router

Severity: Informational

Context: SwapperUniV3.sol#L82-L91, PeripheryPayments.sol#L58-L62

Description: `SwapperUniV3.swap()` performs a swap through Uniswap V3's `SwapRouter` contract by calling `exactInputSingle()`. When swaps are performed through Uniswap V3 routers with `assetIn` with WETH, if an attacker directly transfers sufficient ETH to the router, the swap will be paid from the router's ETH balance instead of from the caller:

```
if (token == WETH9 && address(this).balance >= value) {
    // pay with WETH9
    IWETH9(WETH9).deposit{value: value}(); // wrap only what is needed to pay
    IWETH9(WETH9).transfer(recipient, value);
} else if (payer == address(this)) {
```

In the case of `SwapperUniV3.swap()`, it would leave WETH behind in the contract and a dangling approval to the router. However, there is no way for this be exploited in the protocol's current implementation.

Recommendation: Consider documenting this attack vector as it could be a potential pitfall for future integrations with `SwapperUniV3`.

Collar: Fixed in commit 897a702d by documenting this attack vector.

Spearbit: Verified.

5.3.5 Precision issues with token price representation in `ChainlinkOracle`

Severity: Informational

Context: ChainlinkOracle.sol#L75-L78

Description: In `ChainlinkOracle`, token prices are represented as the price of a unit of base tokens (i.e. `10 ** baseToken.decimals()`) in quote tokens:

```
// feed answer is for base unit (baseUnitAmount), but at feed precision (of feedUnitAmount).
// to translate it to quote precision, we divide by feedUnitAmount and multiply by quoteUnitAmount
// e.g, ETH & USDC using ETH/USD feed (8 decimals): 3000e8 * 1e6 / 1e8 -> 3000e6
return _latestAnswer() * quoteUnitAmount / feedUnitAmount;
```

However, this implementation of calculating the price of one unit of base token in quote tokens (or vice versa) has precision issues when the base/quote token has a low price or decimals. For instance, `currentPrice()` has precision issues when both `_latestAnswer()` and `quoteUnitAmount` are small (i.e. base token has low price and quote token has low decimals).

A concrete example:

- The current price of PEPE is 0.00001735 USD.
- For the PEPE / USD feed, `currentPrice()` is calculated as:
  - `latestAnswer = 0.00001735e18`.
  - `quoteUnitAmount = 1e18`.
  - `feedUnitAmount = 1e18`.
  - `price = 0.00001735e18 * 1e18 / 1e18 = 0.00001735e18`.
- For the USDC / USD feed, `inversePrice()` is calculated as:
  - `latestAnswer = 1e18`.
  - `baseUnitAount = 1e6`.
  - `feedUnitAmount = 1e18`.
  - `price = 1e6 * 1e18 / 1e18 = 1e6`.
- When both feeds are used in `CombinedOracle` to get PEPE / USDC, `currentPrice()` returns:
  - `price1 = 0.00001735e18`, `divisor1 = 1e18`.
  - `price2 = 1e6`.
  - `price = 0.00001735e18 * 1e6 / 1e18 = 17`.
- However, the price of PEPE / USDC without any precision loss is `17.35`.

In the example above, using `ChainlinkOracle` would result in a ~2% deviation in price due to precision loss. If 1,000,000 PEPE was sold an exchange, 17.35 USD would be received in return. In contrast, converting PEPE to USD with `convertToQuoteAmount()` returns `1000_000e18 * 17 / 1e18 = 17e6` USDC, resulting in a loss of 0.35 USD (which is 2% of 17.35).

Similarly, `inversePrice()` experiences the same precision loss if `CombinedOracle` was used to create a USDC / PEPE price feed.

Recommendation: Consider documenting that `ChainlinkOracle` (or any other oracles that represent prices in base / quote) should never be used for tokens with extremely small prices or low decimals.

Collar: Fixed in commit 897a702d by documenting this in more locations and detail than previously.

Spearbit: Verified.

5.3.6 CashAsset(possibly USDC) blacklisted impact on takers, providers and escrow suppliers

Severity: Informational

Context: Global scope; LoansNFT.sol#L490

Description: When an ERC20 `cashAsset` implements a blacklist, it is possible that certain operations within the protocol revert as one of the addresses involved was blacklisted. This is the impact caused if:

- *A provider is blacklisted:*
  - If a provider offer was created, the provider will not be able to cancel it. Could be solved by providing a recipient address to withdraw.
  - The provider will not be able to withdraw from a settle position initially, however he can simply transfer the `CollarProviderNFT` to another non-blacklisted address and then call `withdrawFromSettled` to withdraw.
- *A taker is blacklisted:*
  - Taker can not open/close/roll any loan.
  - `forecloseLoan` can not be called either however escrow owner can still call `lastResortSeizeEscrow`. It is possible in this case that the escrow owner is also blacklisted but in that scenario he can transfer the `EscrowSupplierNFT` to another non-blacklisted address to then call `lastResortSeizeEscrow` successfully.
  - The taker will not be able to withdraw from a settle position initially, however he can simply transfer the `CollarTakerNFT` to another non-blacklisted address and then call `withdrawFromSettled` to withdraw.
- *An escrow supplier is blacklisted:*
  - No impact as the `EscrowSupplierNFT` contract operates with the underlying (WETH, WBTC), not `cashAsset`(USDC).

Recommendation: The main recommendation here would be to consider adding a recipient address to the `CollarProviderNFT.updateOfferAmount` function. This recipient address would receive any withdrawn funds by the provider.

Collar: Fixed in commit 897a702d. As long as the impact is isolated to the blacklisted party only, we don't see this as an impact. It's equivalent to blacklisting hapenning before deposit or after withdrawal. Removing foreclosure (as part of other fixes) removes the only case where a non-blacklisted user is impacted, so we see this as partially resolved.

Spearbit: Agreed on Collar's statement. The removal of the foreclosure function handles the case where a non-blacklisted user is impacted. Moreover, every single situation where a stakeholder is blacklisted can be somehow bypassed by transferring the respective NFT to another non-blacklisted address.

5.3.7 Paused NFTs can't be rescued using `rescueTokens`

Severity: Informational

Context: BaseManaged.sol#L74, BaseNFT.sol#L28

Description: The `_update()` function of `BaseNFT` is being overridden to restrict all transfers when paused. Hence, when `transferFrom()` is called in `resuceTokens()`:

```
IERC721(token).transferFrom(address(this), owner(), amountOrId);
```

It will revert if `token` in the above context is paused and will not lead to a successful rescue for that particular NFT without unpausing (which may possess other situational risks).

Batching `unpause` + `resuceTokens` + `pause` has an implicit dependency on the owner being the same in:

- NFT contracts (which can be unpaused).
- Contracts in which NFTs may need to be rescued from (i.e `onlyOwner` modifier on `rescueTokens`).

That may not hold true always.

Recommendation:

- Document the risk or recovery process stating batching would work by owner being the same.
- Explore if `owner` can be allowed to bypass pause in `_update()`.

Collar: Acknowledged. Since pausing is for dealing with suspected bugs in emergencies, if it's temporary there is no impact, this is because pausing is intended to mitigate risks larger than is caused by it.

If it's not temporary, and both contracts (the host and the rescued token) remain indefinitely paused, they both cannot be unpaused due to a bug. In this case, it's neither possible to allow transferring the buggy token, nor is it necessary, since it's underlying tokens can be rescued.

For example, since Loans holds TakerNFTs, if only taker is briked, there's no need to rescue it (since it's briked and doesn't work), instead Taker's token need to be rescued. OTOH If only Loans is briked, taker is rescuable since is unpaused. Same with Rolls, if Rolls is briked, provider NFT can be rescued, if provider is bricked, Rolls is not, but there's no need to rescue provider tokens.

In the implausible case that paused tokens need to be rescued, as described, ownership of both contracts can be the same (since owner transfer isn't paused), and atomically transferring paused tokens would be possible. We don't think documenting the admin steps for this scenario is needed in the contract, but we can document it when preparing emergency response run books.

Spearbit: Acknowledged.