Collar Protocol core
Security Review

Review by:
Kankodu, Security Researcher

November 21, 2024

Contents

1 Introduction

1.1 Disclaimer

A security review a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While the review endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that a security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.2 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.

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

From Oct 28th to Nov 14th the security researchers conducted a review of collar-protocol-core on commit hash 4343ba39. A total of **10** issues in the following risk categories were identified:

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

3 Findings

3.1 Medium Risk

3.1.1 `_baseURI` Is Not Being Overriden for Any of the Protocol NFTs

Severity: Medium Risk

Context: BaseNFT.sol#L16

Description: An added advantage of representing positions as NFTs is that users can sell their NFTs to exit their positions before expiry. All NFT marketplaces call the `tokenURI(uint256)` method to retrieve the metadata associated with each NFT. Since the protocol is not overriding `function tokenURI(uint256 tokenId) returns (string memory)` or `function _baseURI() returns (string memory)`, the NFT marketplace will be forced to display a default image, which isn't ideal.

Recommendation: To improve this, consider overriding `function _baseURI() returns (string memory)` to return a URI pointing to your a URL, such as `https://nftinfo.collarprotocol.xyz/`, and ensure it returns the correct metadata for each NFT. Alternatively, you could follow the approach of UniswapV3 or SablierV2 and return an SVG directly from the Solidity code. The SVG doesn't need to be as complex or dynamic as UniswapV3's; even a simple, static text like "Collar Position" would be preferable to showing nothing.

Collar: Fixed in commit 91a2c55f.

Kankodu: Fixed.

3.2 Low Risk

3.2.1 Frontrunning Vulnerability in 'updateOfferAmount' That Allows for Double-Spending the Provider Offer

Severity: Low Risk

Context: CollarProviderNFT.sol#L186, EscrowSupplierNFT.sol#L243

Description: The `updateOfferAmount` functionality in both `CollarProviderNFT` and `EscrowNFT` allows for double-spending of the offer amount if a taker frontruns the `updateOfferAmount` transaction.

For example, if a provider has already offered a large amount and wants to increase their offer by a small amount, a taker could frontrun this transaction by accepting the provider's current offer amount. When the provider's transaction is subsequently processed, they will effectively pay the original large amount plus the additional small amount. The taker can then accept this new total amount as well.

A similar issue occurs when the provider wants to decrease the offer by a small amount. Instead of retrieving some of their tokens, the provider, after being frontrun, is forced to maintain an offer amount equal to the previous large amount minus the small amount.

Proof of Concept: Add the following test to `test/unit/CollarProviderNFT.t.sol`:

```
function test_updateOfferAmountIncreaseFrontrunAttack() public {
    // start from an existing offer
    (uint offerId,) = createAndCheckOffer(provider, largeAmount);

    // provider wants to increase offer by a small amount
    uint smallAmount = 1 ether;
    uint newAmount = largeAmount + smallAmount;
    // Likely, provider has given a large approval to providerNFT to avoid multiple approval transactions
    // and has the balance as well
    cashAsset.approve(address(providerNFT), largeAmount + smallAmount);

    assertEq(cashAsset.allowance(provider, address(providerNFT)), largeAmount + smallAmount);
    assertGt(cashAsset.balanceOf(provider), largeAmount + smallAmount);

    uint BIPS_BASE = 10_000;

    uint providerLocked =
        (largeAmount * BIPS_BASE * 365 days) / ((BIPS_BASE * 365 days) + (protocolFeeAPR * duration));

    // someone else frontruns them and takes up the original offer
    vm.startPrank(address(takerNFT));
```

```
    providerNFT.mintFromOffer(offerId, providerLocked, takerNFT.nextPositionId());
    vm.stopPrank();

    CollarProviderNFT.LiquidityOffer memory offer = providerNFT.getOffer(offerId);

    // at this point, the current available amount will be 0
    assertEq(offer.available, 0);

    uint providerBalanceBefore = cashAsset.balanceOf(provider);

    // if the provider's transaction is included now, they will end up offering a large amount + small amount
    // when they simply wanted to increase the original offer by a small amount

    vm.prank(provider);
    providerNFT.updateOfferAmount(offerId, newAmount);

    uint providerBalanceAfter = cashAsset.balanceOf(provider);

    uint difference = providerBalanceBefore - providerBalanceAfter;
    assertEq(difference, largeAmount + smallAmount);

    offer = providerNFT.getOffer(offerId);
    assertEq(offer.available, largeAmount + smallAmount);
}
```

Recommendation: Implement `updateOfferAmount` in a way that prevents this frontrunning vulnerability.

Collar: Added warning in docs for both methods in commit ccc5811f. Decided against code changes due to the following factors:

- Arbitrum and (other L2 deployment destinations) do not expose a public mempool that would be required for exploiting this.
- The ERC-20 issue is known for being a concern that was never exploited.
- Even on L1, not granting excessive approvals is sufficient workaround for any user that may still be concerned about this issue, or FE that is trying to prevent it.
- The likelihood even if were on L1 would be considered very low, because "attacker" has no profit motive other than opening a position larger than "intended"; has only a very narrow time window to do so; only for subset of update calls - non-zero-to-non-zero amounts (instead of just depositing, topping up, or withdrawing), and has excessive approvals; attacker locks their funds for a long duration; attacker puts their funds at risk of loss due to price change; attack is complex (requires frontrunning).

Kankodu: Acknowledged.

3.2.2 Ensure Accurate Event Emission in `OfferUpdated` Event within `mintFromOffer` Function

Severity: Low Risk

Context: CollarProviderNFT.sol#L257

Description: In the `mintFromOffer` function, `msg.sender` represents the taker contract, not the provider. However, when emitting the `OfferUpdated` event, `msg.sender` is incorrectly emitted as the provider.

Recommendation: Update the code to emit the correct provider.

Collar: Fixed in commit 5e93bc67.

Kankodu: Fixed.

3.3 Gas Optimization

3.3.1 Use Custom Errors

Severity: Gas Optimization

Context: CollarProviderNFT.sol#L82

Description: With Solidity version `0.8.27`, support for custom errors in the `require` statement was added. This feature provides a convenient and gas-efficient way to inform users why an operation failed, while also allowing the emission of dynamic information.

Recommendation: Consider replacing string-based errors with custom errors throughout the code.

Collar: We prefer string messages in requires, because custom errors:

- Require ABI to be known to decode the error, limiting UX in wallets, simple front-ends, and limiting composability (errors shown in integrators FE).
- Bloat code with definitions.
- Less readable due to having no spaces.
- Less expressive due to being `NounObjectNamedWithNoSpecialCharacters`.
- Cause visual clutter and distraction in IDE due to being color coded as functions.
- Gas savings compared to strings are negligible on L2.
- Bloat tests due to more verbose expectRevert syntax.

Kankodu: Acknowledged.

3.3.2 Redundant Token Transfer

Severity: Gas Optimization

Context: SwapperUniV3.sol#L85

Description: If the Swapper passes the recipient as `msg.sender` instead of `address(this)`, an additional token transfer can be avoided here.

Recommendation: Save the extra token transfer as suggested.

Collar: Fixed in commit b7f3b3d7.

Kankodu: Fixed.

3.4 Informational

3.4.1 Inconsistency in Events Emitted during Construction

Severity: Informational

Context: CollarProviderNFT.sol#L78

Description: CollaterProviderNFT emits `CollarProviderNFTCreated` to facilitate the indexing of `cashAsset` and `underlying`. `LoanNFT`, `EscrowNFT`, and `Rolls` contracts would also benefit from similar events.

Recommendation: If the event in `CollateralProviderNFT` is redundant, remove it; otherwise, add similar events in the other components for consistency.

Collar: The original reason was a difficulty with identifying the assets used, since they were not emitted by ConfigHub, and instead part of the construction args. We'll look into the need to emit those events as part of more comprehensive subgraph related work.

Kankodu: Acknowledged.

3.4.2 Use Named Constants

Severity: Informational

Context: CollarProviderNFT.sol#L137

Description: Using named constants improves readability. Consider using them consistently throughout the code.

Collar: Fixed in commit 4b4216ff.

Kankodu: Fixed.

3.4.3 Enhance Consistency by Adding Deadlines to `liquidityOffer` in Line with `rollOffer` Requirements

Severity: Informational

Context: CollarProviderNFT.sol#L153

Description: When a provider makes a `liquidityOffer`, there is no deadline for the offer. However, when they make a `rollOffer`, a deadline is required. Adding a `deadline` can be a useful feature compared to offers that don't expire.

Recommendation: Consider adding a `deadline` to the `liquidityOffer` as well.

Collar: Adding the parameter may increase interface clutter without a real UX benefit. The deadline is more useful in roll offers, due to them being short lived, but liquidity offers are intended to be long lived and actively managed.

Kankodu: Acknowledged.

3.4.4 Ensure Consistent Fund Handling

Severity: Informational

Context: Rolls.sol#L164

Description: In `Rolls.createOffer`, the required cash is not pulled from the provider; instead, it happens in `executeRoll` when the taker accepts the offer. `executeRoll` can fail if the provider lacks sufficient balance or approval. In contrast, when the provider creates an offer, funds are pulled from the provider immediately, meaning the cash required for `openPairedPosition` is already in the `providerNFT` contract.

Recommendation: The protocol should be consistent.

Collar: The inconsistency is due to different tradeoffs between capital efficiency (for the provider) and the resistance to spoofing offers in the two cases. Capital efficiency is worse in liquidity offers due to inability to otherwise prevent order spoofing, while in roll offers this is mitigated via the NFT deposit.

Docs have been clarified in commit 6c9cb893.

Kankodu: Acknowledged.

3.4.5 Clarify Possible Discrepancies Between `underlyingAmount` and `loanAmount`

Severity: Informational

Context: LoansNFT.sol#L327

Description: The `underlyingAmount` provided when a loan is created should not be trusted. This is a user input that can be an extremely high or low number and does not necessarily correspond to the resulting loan amount after the swap. A user could sandwich their own transaction to manipulate the price and obtain a high or low resulting loan amount.

Recommendation: It should be clearly communicated to integrators is that `underlyingAmount` and `loanAmount` may not be inline with `TakerPosition.startPrice`

Collar: The amount is pulled from the user, so is considered implicitly "validated". The observation that the position value may not correspond exactly is correct, but this should be made obvious to integrators and users via docs and with basic understanding of the protocol's mechanism. Maintaining the value in

storage is valuable for reference and for ensuring the position was initialized, since all other field values can be 0.

1. Added additional clarifications in commit c8bfa55.
2. Partially mitigated in PR 101 by limiting the difference between underlying and swapped cash values.

Kankodu: Acknowledged.