Rolla contest
Findings & Analysis Report

2022-06-24

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Rolla smart contract system written in Solidity. The audit contest took place between March 17—March 24 2022.

Wardens

23 Wardens contributed reports to the Rolla contest:

1. WatchPug (jtp and ming)
2. rayn
3. hyh
4. IllIllI
5. 0xDjango
6. jayjonah8
7. 0xmint
8. 0x1f8b
9. cccz
10. danb
11. berndartmueller
12. Ruhum
13. 0xkatana
14. gzeon
15. defsec
16. badbird
17. cryptphi
18. robee
19. TerrierLover
20. 0v3rf10w

21. remora
22. Jujic

This contest was judged by Alberto Cuesta Cañada.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 14 unique vulnerabilities. Of these vulnerabilities, 4 received a risk rating in the category of HIGH severity and 10 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 20 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Rolla contest repository, and is composed of 24 smart contracts written in the Solidity programming language and includes 4,071 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic

- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](https://code4rena.com).

High Risk Findings (4)

[H-01] Incorrect strike price displayed in name/symbol of qToken

Submitted by rayn

`_slice()` in `options/QTokenStringUtils.sol` cut a string into `string[start:end]` However, while fetching bytes, it uses `bytes(_s)[_start+1]` instead of `bytes(_s)[_start+i]`. This causes the return string to be composed of `_s[start]*(_end-_start)`. The result of this function is then used to represent the decimal part of strike price in name/symbol of qToken, leading to potential confusion over the actual value of options.

Proof of Concept

ERC20 tokens are usually identified by their name and symbol. If the symbols are incorrect, confusions may occur. Some may argue that even if names and symbols are not accurate, it is still possible to identify correct information/usage of tokens by querying the provided view functions and looking at its interactions with other contracts. However, the truth is many users of those tokens are not very tech savvy, and it is reasonable to believe a large proportion of users are not equipped with enough knowledge, or not willing to dig further than the plain symbols and names. This highlights the importance of maintaining a correct facade for ERC20 tokens.

The bug demonstrated here shows that any qToken with decimals in its strike price will be misdisplayed, and the maximal difference between actual price and displayed one can be up to 0.1 BUSD.

The exploit can be outlined through the following steps:

- Alice created a call option with strike price 10000.90001. The expected symbol should for this qToken should be : `ROLLA WETH 31-December-2022 10000.90001 Call`
- Both `_qTokenName()` and `_qTokenSymbol()` in `options/QTokenStringUtils.sol` use `_displayedStrikePrice()` to get the strike price string which should be `10000.90001`

https://github.com/RollaProject/quant-protocol/blob/98639a3/contracts/options/QTokenStringUtils.sol#L38
https://github.com/RollaProject/quant-protocol/blob/98639a3/contracts/options/QTokenStringUtils.sol#L90

```
    function _qTokenName(
        address _quantConfig,
        address _underlyingAsset,
        address _strikeAsset,
        uint256 _strikePrice,
        uint256 _expiryTime,
        bool _isCall
    ) internal view virtual returns (string memory tokenName) {
        string memory underlying = _assetSymbol(_quantConfig, _u
        string memory displayStrikePrice = _displayedStrikePrice
            _strikePrice,
            _strikeAsset
        );

        ...

        tokenName = string(
            abi.encodePacked(
                "ROLLA",
                " ",
                underlying,
                " ",
                _uintToChars(day),
                "-",
                monthFull,
                "-",
                Strings.toString(year),
                " ",
                displayStrikePrice,
                " ",
                typeFull
            )
        );
    }



    function _qTokenSymbol(
        address _quantConfig,
        address _underlyingAsset,
        address _strikeAsset,
        uint256 _strikePrice,
        uint256 _expiryTime,
        bool _isCall
    ) internal view virtual returns (string memory tokenSymbol)
        string memory underlying = _assetSymbol(_quantConfig, _u
        string memory displayStrikePrice = _displayedStrikePrice
            _strikePrice,
            _strikeAsset
        );

        // convert the expiry to a readable string
        (uint256 year, uint256 month, uint256 day) = DateTime.ti
            _expiryTime
        );

        // get option type string
        (string memory typeSymbol, ) = _getOptionType(_isCall);

        // get option month string
        (string memory monthSymbol, ) = _getMonth(month);

        /// concatenated symbol string
        tokenSymbol = string(
            abi.encodePacked(
                "ROLLA",
                "-",
                underlying,
                "-",
                _uintToChars(day),
                monthSymbol,
                _uintToChars(year),
                "-",
                displayStrikePrice,
                "-",
                typeSymbol
            )
        );
    }
```

- `_displayedStrikePrice()` combines the quotient and the remainder to form the strike price string. The remainder use `_slice` to compute. In this case, the quotient is `10000` and the remainder is `90001`

https://github.com/RollaProject/quant-protocol/blob/98639a3/contracts/options/QTokenStringUtils.sol#L136

```
    function _displayedStrikePrice(uint256 _strikePrice, address _st
        internal
        view
        virtual
        returns (string memory)
    {
        uint256 strikePriceDigits = ERC20(_strikeAsset).decimals
        uint256 strikePriceScale = 10**strikePriceDigits;
        uint256 remainder = _strikePrice % strikePriceScale;
        uint256 quotient = _strikePrice / strikePriceScale;
        string memory quotientStr = Strings.toString(quotient);

        if (remainder == 0) {
            return quotientStr;
        }

        uint256 trailingZeroes;
        while (remainder % 10 == 0) {
            remainder /= 10;
            trailingZeroes++;
        }

        // pad the number with "1 + starting zeroes"
        remainder += 10**(strikePriceDigits - trailingZeroes);

        string memory tmp = Strings.toString(remainder);
        tmp = _slice(tmp, 1, (1 + strikePriceDigits) - trailingZ

        return string(abi.encodePacked(quotientStr, ".", tmp));
    }
```

- However inside the loop of `_slice()`, `slice[i] = bytes(_s)[_start + 1];` lead to an incorrect string, which is `90001`

https://github.com/RollaProject/quant-protocol/blob/98639a3/contracts/options/QTokenStringUtils.sol#L206

```
    function _slice(
        string memory _s,
        uint256 _start,
        uint256 _end
    ) internal pure virtual returns (string memory) {
        uint256 range = _end - _start;
        bytes memory slice = new bytes(range);
        for (uint256 i = 0; i < range; ) {
            slice[i] = bytes(_s)[_start + 1];
            unchecked {
                ++i;
            }
        }

        return string(slice);
    }
```

- The final qtoken name now becomes `ROLLA WETH 31-December-2022 10000.99999 Call`, which results in confusion over the actual value of options.

Recommended Mitigation Steps

Fix the bug in the `_slice()`

```
    function _slice(
        string memory _s,
        uint256 _start,
        uint256 _end
    ) internal pure virtual returns (string memory) {
        uint256 range = _end - _start;
        bytes memory slice = new bytes(range);
        for (uint256 i = 0; i < range; ) {
            slice[i] = bytes(_s)[_start + i];
            unchecked {
                ++i;
            }
        }

        return string(slice);
    }
```

[0xca11 (Rolla) confirmed, resolved, and commented]():

> Resolved in [RollaProject/quant-protocol#77]()

[H-02] Mint spread collateral-less and conjuring collateral claims out of thin air with implicit arithmetic rounding and flawed int to uint conversion

Submitted by rayn

[QuantMath.sol#L137]()
[QuantMath.sol#L151]()
[SignedConverter.sol#L28]()

This report presents 2 different incorrect behaviour that can affect the correctness of math calculations:

1. Unattended Implicit rounding in QuantMath.sol `div` and `mul`
2. Inappropriate method of casting integer to unsigned integer in SignedConverter.sol `intToUint`

Bug 1 affects the correctness when calculating collateral required for `_mintSpread`. Bug 2 expands the attack surface and allows attackers to target the `_claimCollateral` phase instead. Both attacks may result in tokens being stolen from Controller in the worst case, but is most likely too costly to exploit under current BNB chain environment. The potential impact however, should not be taken lightly, since it is known that the ethereum environment in highly volatile and minor changes in the environment can suddenly make those bugs cheap to exploit.

Proof of Concept

In this section, we will first present bug 1, and then demonstrate how this bug can be exploited. Then we will discuss how bug 2 opens up more attack chances and go over another PoC.

Before getting started, we should go over an important concept while dealing with fixed point number — rounding. Math has no limits on precision, but computers do. This problem is especially critical to systems handling large amount of "money" that is allowed to be arbitrarily divided. A common way for ethereum smart contract developers to handle this is through rounding numbers. Rolla is no exception.

In QuantMath, Rolla explicitly wrote the `toScaledUint` function to differentiate between rounding numbers up or down when scaling numbers to different precision (or we call it `_decimals` here). The intended usage is to scale calculated numbers (amount of tokens) up when Controller is the receiver, and scale it down when Controller is sender. In theory, this function should guarantee Controller can never "lose tokens" due to rounding.

```
library QuantMath {
    ...
    struct FixedPointInt {
        int256 value;
    }

    int256 private constant _SCALING_FACTOR = 1e27;
    uint256 private constant _BASE_DECIMALS = 27;

    ...

    function toScaledUint(
        FixedPointInt memory _a,
        uint256 _decimals,
        bool _roundDown
    ) internal pure returns (uint256) {
        uint256 scaledUint;

        if (_decimals == _BASE_DECIMALS) {
            scaledUint = _a.value.intToUint();
        } else if (_decimals > _BASE_DECIMALS) {
            uint256 exp = _decimals - _BASE_DECIMALS;
            scaledUint = (_a.value).intToUint() * 10**exp;
        } else {
            uint256 exp = _BASE_DECIMALS - _decimals;
            uint256 tailing;
            if (!_roundDown) {
                uint256 remainer = (_a.value).intToUint() % 10**
                if (remainer > 0) tailing = 1;
            }
            scaledUint = (_a.value).intToUint() / 10**exp + tail
        }

        return scaledUint;
    }
    ...
}
```

In practice, the above function also works quite well (sadly, not perfect, notice the `intToUint` function within. We will come back to this later), but it only works if we can promise that before entering this function, all numbers retain full precision and is not already rounded. This is where `div` and `mul` comes into play. As we can easily see in the snippet below, both functions involve the division operator '/', which by default discards the decimal part of the calculated result (be aware to not confuse this with the `_decimal` used while scaling FixedPointInt). The operation here results in an implicit round down, which limits the effectiveness of explicit rounding in `toScaledUint` showned above.

```
    function mul(FixedPointInt memory a, FixedPointInt memory b)
        internal
        pure
        returns (FixedPointInt memory)
    {
        return FixedPointInt((a.value * b.value) / _SCALING_FACT
    }


    function div(FixedPointInt memory a, FixedPointInt memory b)
        internal
        pure
        returns (FixedPointInt memory)
    {
        return FixedPointInt((a.value * _SCALING_FACTOR) / b.val
    }
```

Now let's see how this implicit rounding can causes troubles. We start with the `_mintSpread` procedure creating a call credit spread. For brevity, the related code is not shown, but here's a summary of what is done.

- `Controller._mintSpread`
    - `QuantCalculator.getCollateralRequirement`
        - `FundsCalculator.getCollateralRequirement`
            - `FundsCalculator.getOptionCollateralRequirement`
                - `FundsCalculator.getCallCollateralRequirement`
                    - scales `_qTokenToMintStrikePrice` from `_strikeAssetDecimals (8)` to `_BASE_DECIMALS (27)`
                    - scales `_qTokenForCollateralStrikePrice` from `_strikeAssetDecimals (8)` to `_BASE_DECIMALS (27)`
                    - `collateralPerOption = (collateralStrikePrice.sub(mintStrikePrice)).div(collateralStrikePrice)`
                - scale `_optionsAmount` from `_optionsDecimals (18)` to `_BASE_DECIMALS (27)`
                - `collateralAmount = _optionsAmount.mul(collateralPerOption)`
            - uses `qTokenToMint.underlyingAsset` (weth or wbtc) as collateral
        - scale and round up `collateralAmountFP` from `_BASE_DECIMALS (27)` to `payoutDecimals (18)`

If we extract all the math related stuff, it would be something like below

```
def callCreditSpreadCollateralRequirement(_qTokenToMintStrikePri
        X1 = _qTokenToMintStrikePrice * 10^19
        X2 = _qTokenForCollateralStrikePrice * 10^19
        X3 = _optionsAmount * 10^9

        assert X1 < X2          #credit spread

        Y1 = (X2 - X1) * 10^27 // X2    #implicit round down due
        Y2 = Y1 * X3 // 10^27   #implicit round down due to mul

        Z = Y2 // 10^9
        if Y2 % 10^9 > 0:       #round up since we are minting s
                Z+=1
        return Z
```

Both implicit round downs can be abused, but we shall focus on the `mul` one here. Assume we follow the following actions

1. create option `A` with strike price `10 + 10^-8 BUSD (10^9 + 1 under 8 decimals) <-> 1 WETH`
2. create option `B` with strike price `10 BUSD (10^9 under 8 decimals) <-> 1 WETH`
3. mint `10^-18` (1 under 18 decimals) option `A`
   3-1. `pay 1 eth`
4. mint `10^-18` (1 under 18 decimals) spread `B` with `A` as collateral
   4-1. `X1 = _qTokenToMintStrikePrice * 10^19 = 10^9 * 10^19 = 10^28`
   4-2. `X2 = _qTokenToMintStrikePrice * 10^19 = (10^9 + 1) * 10^19 = 10^28 + 10^19`
   4-3. `X3 = _optionsAmount * 10^9 = 1 * 10^9 = 10^9`
   4-4. `Y1 = (X2 - X1) * 10^27 // X2 = (10^28 + 10^19 - 10^28) * 10^27 // (10^28 + 10^19) = 99999999000000000`
   4-5. `Y2 = Y1 * X3 // 10^27 = 99999999000000000 * 10^9 / 10^27 = 0`
   4-6. `Z = Y2 // 10^9 = 0`
   4-7. `Y2 % 10^9 = 0` so `Z` remains unchanged

We minted a call credit spread without paying any fee.

Now let's think about how to extract the value we conjured out of thin air. To be able to withdraw excessive collateral, we can choose to do a excercise+claim or neutralize current options. Here we take the neutralize path.

For neutralizing spreads, the procedure is basically the same as minting spreads, except that the explicit round down is taken since `Controller` is the payer here. The neutralize procedure returns the `qToken` used as collateral and pays the collateral fee back. The math part can be summarized as below.

```
def neutralizeCreditSpreadCollateralRequirement(_qTokenToMintStr
        X1 = _qTokenToMintStrikePrice * 10^19
        X2 = _qTokenForCollateralStrikePrice * 10^19
        X3 = _optionsAmount * 10^9

        assert X1 < X2          #credit spread

        Y1 = (X2 - X1) * 10^27 // X2    #implicit round down due
        Y2 = Y1 * X3 // 10^27   #implicit round down due to mul

        Z = Y2 // 10^9  #explicit scaling
        return Z
```

There are two challenges that need to be bypassed, the first one is to avoid implicit round down in `mul`, and the second is to ensure the revenue is not rounded away during explicit scaling. To achieve this, we first mint `10^-9 + 2 * 10^-18` spreads seperately (10^9 + 2 under 18 decimals), and as shown before, no additional fees are required while minting spread from original option. Then we neutralize all those spreads at once, the calculation is shown below.

1. neutralize `10^-9 + 2 * 10^-18` (10^9 + 2 under 18 decimals) spread `B`
    4-1. `X1 = _qTokenToMintStrikePrice * 10^19 = 10^9 * 10^19 = 10^28`
    4-2. `X2 = _qTokenToMintStrikePrice * 10^19 = (10^9 + 1) * 10^19 = 10^28 + 10^19`
    4-3. `X3 = _optionsAmount * 10^9 = (10^9 + 2) * 10^9 = 10^18 + 2`
    4-4. `Y1 = (X2 - X1) * 10^27 // X2 = (10^28 + 10^19 - 10^28) * 10^27 // (10^28 + 10^19) = 999999999000000000`
    4-5. `Y2 = Y1 * X3 // 10^27 = 999999999000000000 * (10^18 + 2) / 10^27 = 1000000000`
    4-6. `Z = Y2 // 10^9 = 10^9 // 10^9 = 1`

And with this, we managed to generate 10^-18 weth of revenue.

This approach is pretty impractical due to the requirement of minting 10^-18 for `10^9 + 2` times. This montrous count mostly likely requires a lot of gas to pull off, and offsets the marginal revenue generated through our attack. This leads us to explore other possible methods to bypass this limitation.

It’s time to start looking at the second bug.

Recall we mentioned the second bug is in `intToUint`, so here’s the implementation of it. It is not hard to see that this is actually an `abs` function named as `intToUint`.

```
function intToUint(int256 a) internal pure returns (uint256)
    if (a < 0) {
        return uint256(-a);
    } else {
        return uint256(a);
    }
}
```

Where is this function used? And yes, you guessed it, in `QuantCalculator.calculateClaimableCollateral`. The process of claiming collateral is quite complex, but we will only look at the specific case relevant to the exploit. Before reading code, let’s first show the desired scenario. Note that while we wait for expiry, there are no need to sell any option/spread.

1. mint a `qTokenLong` option
2. mint a `qTokenShort` spread with `qTokenLong` as collateral
3. wait until expire, and expect expiryPrice to be between qTokenLong and qTokenShort

```
----------- qTokenLong strike price

----------- expiryPrice

----------- qTokenShort strike price
```

Here is the outline of the long waited claimCollateral for spread.

- `Controller._claimCollateral`
    - `QuantCalculator.calculateClaimableCollateral`
        - `FundsCalculator.getSettlementPriceWithDecimals`
        - `FundsCalculator.getPayout` for qTokenLong
            - qTokenLong strike price is above expiry price, worth 0
        - `FundsCalculator.getCollateralRequirement`
            - This part we saw earlier, omit details
        - `FundsCalculator.getPayout` for qTokenShort
            - uses `qTokenToMint.underlyingAsset` (weth or wbtc) as collateral
            - `FundsCalculator.getPayoutAmount` for qTokenShort
                - scale `_strikePrice` from `_strikeAssetDecimals (8)` to `_BASE_DECIMALS (27)`
                - scale `_expiryPrice.price` from `_expiryPrice.decimals (8)` to `_BASE_DECIMALS (27)`
                - scale `_amount` from `_optionsDecimals (18)` to `_BASE_DECIMALS (27)`
                - `FundsCalculator.getPayoutForCall` for qTokenShort
                    - `payoutAmount = expiryPrice.sub(strikePrice).mul(amount).div(expiryPrice)`
        - `returnableCollateral = payoutFromLong.add(collateralRequirement).sub(payoutFromShort)`

- scale and round down `abs(returnableCollateral)` from `_BASE_DECIMALS (27)` to `payoutDecimals (18)`

Again, we summarize the math part into a function.

```
    def claimableCollateralCallCreditSpreadExpiryInbetween(_qTokenSh

            def callCreditSpreadCollateralRequirement(_qTokenToMintS
                    X1 = _qTokenToMintStrikePrice * 10^19
                    X2 = _qTokenForCollateralStrikePrice * 10^19
                    X3 = _optionsAmount * 10^9

                    Y1 = (X2 - X1) * 10^27 // X2
                    Y2 = Y1 * X3 // 10^27
                    return Y2

            def callCreditSpreadQTokenShortPayout(_strikePrice, _exp
                    X1 = _strikePrice * 10^19
                    X2 = _expiryPrice * 10^19
                    X3 = _amount * 10^9

                    Y1 = (X2-X1) * X3 // 10^27
                    Y2 = Y1 * 10^27 // X2
                    return Y2


            assert _qTokenShortStrikePrice > _expiryPrice > _qTokenI

            A1 = payoutFromLong = 0
            A2 = collateralRequirement = callCreditSpreadCollateralR
            A3 = payoutFromShort = callCreditSpreadQTokenShortPayout

            B1 = A1 + A2 - A3

            Z = abs(B1) // 10^9
            return Z
```

Given the context, it should be pretty easy to imagine what I am aiming here, to make `B1 < 0`. We already know `A1 = 0`, so the gaol basically boils down to making `A2 < A3`. Let's further simplify this requirement and see if the equation is solvable.

```
X = _qTokenLongStrikePrice (8 decimals)
Y = _expiryPrice (8 decimals)
Z = _qTokenShortStrikePrice (8 decimals)
A = _amount (scaled to 27 decimals)

assert X>Y>Z>0
assert X,Y,Z are integers
assert (((X - Z) * 10^27 // X) * A // 10^27) < (((Y - Z) * A //
```

Notice apart from the use of `X` and `Y`, the two sides of the equation only differs by when `A` is mixed into the equation, meaning that if we temporarily ignore the limitation and set `X = Y`, as long as left hand side of equation does an implicit rounding after dividing by X, right hand side will most likely be larger.

Utilizing this, we turn to solve the equation of:

```
(X-Z) / X - (Y-Z) / Y < 10^-27
=> Z / Y - Z / X < 10^-27
=> (Z = 1 yields best solution)
=> 1 / Y - 1 / X < 10^-27
=> X - Y < X * Y * 10^-27
=> 0 < X * Y - 10^27 * X + 10^27 * Y

=> require X > Y, so model Y as X - B, where B > 0 and B is an i
=> 0 < X^2 - B * X - 10^27 * B
```

It is not easy to see that the larger `X` is, the larger the range of allowed `B`. This is pretty important since `B` stands for the range of expiry prices where attack could work, so the larger it is, the less accurate our guess can be to profit.

Apart form range of `B`, value of `X` is the long strike price and upper bound of range `B`, so we would also care about it, a simple estimation shows that `X` must be above `10^13.5 (8 decimals)` for there to be a solution, which amounts to about `316228 BUSD <-> 1 WETH`. This is an extremely high price, but not high enough to be concluded as unreachable in the near future. So let’s take a slightly generous number of `10^14 - 1` as X and calculate the revenue generated following this exploit path.

```
0 < (10^14 - 1)^2 - B * (10^14 - 1) - 10^27 * B
=> (10^14 - 1)^2 / (10^14 - 1 + 10^27) > B
=> B <= 9
```

Now we've got the range of profitable expiry price. As we concluded earlier, the range is extremely small with a modest long strike price, but let's settle with this for now and see how much profit can be generated if we get lucky. To calculate profit, we take `_qTokenLongStrikePrice = 10^14 - 1 (8 decimals)`, `_qTokenShortStrikePrice = 1 (8 decimals)`, `_expiryPrice = 10^14 - 2 (8 decimals)` and `_amount = 10^28 (18 decimals)` and plug it back into the function.

1. in `callCreditSpreadCollateralRequirement`
   1-1. `X1 = _qTokenForCollateralStrikePrice * 10^19 = 1 * 10^19 = 10^19`
   1-2. `X2 = _qTokenToMintStrikePrice * 10^19 = (10^14 - 1) * 10^19 = 10^33 - 10^19`
   1-3. `X3 = _optionsAmount * 10^9 = 10^28 * 10^9 = 10^37`
   1-4. `Y1 = (X2 - X1) * 10^27 // X2 = (10^33 - 2 * 10^19) * 10^27 // (10^33 - 10^19) = 999999999999998999999999999`
   1-5. `Y2 = Y1 * X3 // 10^27 = 999999999999998999999999999 * 10^37 // 10^27 = 999999999999998999999999999 * 10^10`
2. in `callCreditSpreadQTokenShortPayout`
   2-1. `X1 = _strikePrice * 10^19 = 1 * 10^19 = 10^19`
   2-2. `X2 = _expiryPrice * 10^19 = (10^14 - 2) * 10^19 = 10^33 - 2 * 10^19`
   2-3. `X3 = _amount * 10^9 = 10^28 * 10^9 = 10^37`
   2-4. `Y1 = (X2 - X1) * X3 // 10^27 = (10^33 - 3 * 10^19) * 10^37 // 10^27 = 99999999999997 * 10^29`
   2-5. `Y2 = Y1 * 10^27 / X2 = (99999999999997 * 10^28) * 10^27 / (10^33 - 2 * 10^19) = 9999999999999899999999999997999999999
3. combine terms
   3-1. `B1 = A1 + A2 - A3 = 0 + 9999999999999899999999999990000000000 - 9999999999999899999999999997999999999 = -2000000001<br> 3-2.` Z = abs(B1) // 10^9 = 2000000000 // 10^9 = 2

And with this, we managed to squeeze 2 wei from a presumably worthless collateral.

This attack still suffers from several problems

1. cost of WETH in BUSD is way higher than current market
2. need to predict target price accurately to profit
3. requires large amount of WETH to profit

While it is still pretty hard to pull off attack, the requirements seems pretty more likely to be achievable compared to the first version of exploit. Apart from this, there is also the nice property that this attack allows profit to scale with money invested.

This concludes our demonstration of two attacks against the potential flaws in number handling.

Tools Used

vim, ganache-cli

Recommended Mitigation Steps

For `div` and `mul`, adding in a similar opt-out round up argument would work. This would require some refactoring of code, but is the only way to fundamentally solve the problem.

For `intToUint`, I still can't understand what the original motive is to design it as `abs` in disguise. Since nowhere in this project would we benefit from the current `abs` behaviour, in my opinion, it would be best to adopt a similar strategy to the `uintToInt` function. If the value goes out of directly convertable range (< 0), revert and throw an error message.

0xca11 (Rolla) confirmed, resolved, and commented:

> Resolved by adding explicit rounding on fixed-point multiplication and division operations: RollaProject/quant-protocol#91.

[H-03] Wrong implementation of `EIP712MetaTransaction`

Submitted by WatchPug

1. `EIP712MetaTransaction` is a utils contract that intended to be inherited by concrete (actual) contracts, therefore. it's initializer function should not use the `initializer` modifier, instead, it should use `onlyInitializing` modifier. See the implementation of openzeppelin `EIP712Upgradeable` initializer function.

EIP712MetaTransaction.sol#L102-L114

```
/// @notice initialize method for EIP712Upgradeable
/// @dev called once after initial deployment and every upgr
/// @param _name the user readable name of the signing domai
/// @param _version the current major version of the signinc
function initializeEIP712(string memory _name, string memory
    public
    initializer
{
    name = _name;
    version = _version;

    __EIP712_init(_name, _version);
}
```

Otherwise, when the concrete contract's initializer function (with a `initializer` modifier) is calling EIP712MetaTransaction's initializer function, it will be mistok as reentered and so that it will be reverted (unless in the context of a constructor, e.g. Using @openzeppelin/hardhat-upgrades `deployProxy()` to initialize).

https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/v4.5.1/contracts/proxy/utils/Initializable.sol#L50-L53

```
/**
 * @dev Modifier to protect an initializer function from bei
 */
modifier initializer() {
    // If the contract is initializing we ignore whether _ir
    // inheritance patterns, but we only do this in the cont
    // contract may have been reentered.
    require(_initializing ? _isConstructor() : !_initializec
    bool isTopLevelCall = !_initializing;
    if (isTopLevelCall) {
        _initializing = true;
        _initialized = true;
    }

    _;

    if (isTopLevelCall) {
        _initializing = false;
    }
}
```

See also: https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/releases/tag/v4.4.1

2. `initializer` can only be called once, it can not be “called once after every upgrade”.

EIP712MetaTransaction.sol#L102-L114

```
    /// @notice initialize method for EIP712Upgradeable
    /// @dev called once after initial deployment and every upgr
    /// @param _name the user readable name of the signing domai
    /// @param _version the current major version of the signing
    function initializeEIP712(string memory _name, string memory
        public
        initializer
    {
        name = _name;
        version = _version;

        __EIP712_init(_name, _version);
    }
```

3. A utils contract that is not expected to be deployed as a standalone contract should be declared as `abstract`. It’s `initializer` function should be `internal`.

See the implementation of openzeppelin `EIP712Upgradeable`.

https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/v4.5.1/contracts/utils/cryptography/draft-EIP712Upgradeable.sol#L28

```
abstract contract EIP712Upgradeable is Initializable {
    // ...
}
```

Recommended Mitigation Steps

Change to:

```
abstract contract EIP712MetaTransaction is EIP712Upgradeable {
    // ...
}
```

```
    /// @notice initialize method for EIP712Upgradeable
    /// @dev called once after initial deployment.
    /// @param _name the user readable name of the signing domai
    /// @param _version the current major version of the signinc
    function __EIP712MetaTransaction_init(string memory _name, s
        internal
        onlyInitializing
    {
        name = _name;
        version = _version;

        __EIP712_init(_name, _version);
    }
```

0xca11 (Rolla) confirmed, resolved, and commented:

> Resolved in RollaProject/quant-protocol@25112fa, but upgradeability was later removed as per RollaProject/quant-protocol#90.

[H-04]

`EIP712MetaTransaction.executeMetaTransaction()` failed txs are open to replay attacks

Submitted by WatchPug

Any transactions that fail based on some conditions that may change in the future are not safe to be executed again later (e.g. transactions that are based on others actions, or time-dependent etc).

In the current implementation, once the low-level call is failed, the whole tx will be reverted and so that `_nonces[metaAction.from]` will remain unchanged.

As a result, the same tx can be replayed by anyone, using the same signature.

EIP712MetaTransaction.sol#L86

```
    function executeMetaTransaction(
        MetaAction memory metaAction,
        bytes32 r,
        bytes32 s,
        uint8 v
    ) external payable returns (bytes memory) {
        require(
            _verify(metaAction.from, metaAction, r, s, v),
            "signer and signature don't match"
        );

        uint256 currentNonce = _nonces[metaAction.from];

        // intentionally allow this to overflow to save gas,
        // and it's impossible for someone to do 2 ^ 256 - 1 met
        unchecked {
            _nonces[metaAction.from] = currentNonce + 1;
        }

        // Append the metaAction.from at the end so that it can
        // from the calling context (see _msgSender() below)
        (bool success, bytes memory returnData) = address(this).
            abi.encodePacked(
                abi.encodeWithSelector(
                    IController(address(this)).operate.selector,
                    metaAction.actions
                ),
                metaAction.from
            )
        );

        require(success, "unsuccessful function call");
        emit MetaTransactionExecuted(
            metaAction.from,
            payable(msg.sender),
            currentNonce
        );
        return returnData;
    }
```

See also the implementation of OpenZeppelin's `MinimalForwarder`:

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.5.0/contracts/metatx/MinimalForwarder.sol#L42-L66

Proof of Concept

Given:

- The collateral is USDC;
- Alice got `10,000 USDC` in the wallet.
- Alice submitted a MetaTransaction to `operate()` and `_mintOptionsPosition()` with `10,000 USDC`;
- Before the MetaTransaction get executed, Alice sent `1,000 USDC` to Bob;
- The MetaTransaction submited by Alice in step 1 get executed but failed;
- A few days later, Bob sent `1,000 USDC` to Alice;
- The attacker can replay the MetaTransaction failed to execute at step 3 and succeed.

Alice's `10,000 USDC` is now been spent unexpectedly against her will and can potentially cause fund loss depends on the market situation.

Recommended Mitigation Steps

Failed txs should still increase the nonce.

While implementating the change above, consider adding one more check to require sufficient gas to be paid, to prevent “insufficient gas griefing attack” as described in this article.

0xca11 (Rolla) confirmed, resolved, and commented:

> Meta transactions replay and insufficient gas griefing attacks are now prevented since RollaProject/quant-protocol#80.

Medium Risk Findings (10)

[M-01] No use of upgradeable SafeERC20 contract in `Controller.sol`

Submitted by jayjonah8

Controller.sol makes use of Open Zeppelins `ReentrancyGuardUpgradeable.sol` in the file but does not use an upgradeable version of SafeERC20.sol

Proof of Concept

Controller.sol#L5

Recommended Mitigation Steps

Make use of Open Zeppelins upgradeable version of the SafeERC20.sol contract.

0xca11 (Rolla) resolved and commented:

> Resolved in RollaProject/quant-protocol#76.

[M-02] `COLLATERAL_MINTER_ROLE` can be granted by the

deployer of `QuantConfig` and mint arbitrary amount of tokens

Submitted by cccz, also found by danb, and WatchPug

```
        function mintCollateralToken(
            address recipient,
            uint256 collateralTokenId,
            uint256 amount
        ) external override {
            require(
                quantConfig.hasRole(
                    quantConfig.quantRoles("COLLATERAL_MINTER_ROLE")
                    msg.sender
                ),
                "CollateralToken: Only a collateral minter can mint
            );

            emit CollateralTokenMinted(recipient, collateralTokenId,

            _mint(recipient, collateralTokenId, amount, "");
        }
```

Using the mintCollateralToken() function of CollateralToken, an address with COLLATERAL*MINTER*ROLE can mint an arbitrary amount of tokens.

If the private key of the deployer or an address with the COLLATERAL*MINTER*ROLE is compromised, the attacker will be able to mint an unlimited amount of collateral tokens.

We believe this is unnecessary and poses a serious centralization risk.

Proof of Concept

CollateralToken.sol#L101-L117
CollateralToken.sol#L138-L160

Recommended Mitigation Steps

Consider removing the COLLATERAL*MINTER*ROLE, make the CollateralToken only mintable by the owner, and make the Controller contract to be the owner and therefore the only minter.

0xca11 (Rolla) confirmed

alcueca (judge) commented:

> Per sponsor comment on #47:

> “The roles are renounced as per our deployment config covered in the docs. But this bug is still valid as the role OPTIONS*MINTER*ROLE can be reassigned”.

> Taking this one as main, with the vulnerability being that several of the MINTER and BURNER roles can be reassigned and have unnecessary powers that can be used to rug users.

0xca11 (Rolla) resolved and commented:

> All roles were removed from the protocol, and now only the Controller contract can mint QTokens and CollateralTokens.

> RollaProject/quant-protocol#90

[M-03] Usage of deprecated Chainlink functions

Submitted by Ruhum, also found by 0x1f8b, cccz, and WatchPug

The Chainlink functions `latestAnswer()` and `getAnswer()` are deprecated. Instead, use the `latestRoundData()` and `getRoundData()` functions.

Proof of Concept

ChainlinkOracleManager.sol#L120

ChainlinkFixedTimeOracleManager.sol#L81

ChainlinkFixedTimeOracleManager.sol#L84

Go to https://etherscan.io/address/0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419#code and search for `latestAnswer()` or `getAnswer()`. You'll find the deprecation notice.

Recommended Mitigation Steps

Switch to `latestRoundData()` as described here.

0xca11 (Rolla) confirmed, resolved, and commented:

> Fixed in RollaProject/quant-protocol#89.

[M-04] `ConfigTimeLockController` will put `QuantConfig` in a stalemate (rendering it unusable)

Submitted by 0xmint

The QuantConfig contract has these important setters, setProtocolAddress(), setProtocolUint256, setProtocolBoolean() and setProtocolRole(). This contract is subjected to a timelock before all such processes above are executed. But, the issue arises in the fact that in configTimeLockController, the state variable minimum delay can be set to an arbitrary value, up to type(uint256).max(cannot assume what value will be set) and could potentially render the QuantConfig contract unusable . All the previous values and addresses would not be able to be changed because of a very high delay being set:

ConfigTimelockController.sol#L28

I discussed with one of the devs about the use of this specific mapping :

QuantConfig.sol#L27

After discussions with one of the devs(#0xca11.eth) , it was understood that these values are for the rollaOrderFee which is a part of their limit order protocol contract(outside of the scope of the contest) but given the argument above, its configuration will be severely impacted (old percentage fees won't be able to be changed).Rolla limit order protocol depends on this configuration setting within QuantConfig.

Recommended Mitigation Steps

It is recommended that a constant be declared with a MAXIMUM_DELAY and whatever 'minimum delay' that is set thereafter should be below this value since there's another function setDelay () which can also be of high arbitrary value:

```
require(minimum delay ≤MAXIMUM_DELAY, " too high")
```

0xca11 (Rolla) confirmed, resolved, and commented:

> Both the ConfigTimeLockController and QuantConfig contracts were removed from the protocol.

> RollaProject/quant-protocol#90

[M-05] QTokens with the same symbol will lead to mistakes

Submitted by IllIllI

The `README.md` states:

> Bob can then trade the QToken with Alice for a premium. The method for doing that is beyond the scope of the protocol but can be done via any smart contract trading platform e.g. 0x.

It is therefore important that tokens be easily identifiable so that trading on DEXes is not error-prone.

Impact

Currently the `QToken name` includes the full year but the `QToken` symbol only contains the last two digits of the year, which can lead to mistakes. If someone mints a QToken with an expiry 100 years into the future, then the year will be truncated and appear as if the token expired this year. Normal centralized exchanges prevent this by listing options themselves and ensuring that there are never two options with the same identifier at the same time. The Rolla protocol does not have any such protections. Users must be told to not only check that the symbol name is what they expect, but to also separately check the token name or the specific expiry, or they might buy the wrong option on a DEX, or have fat-fingered during minting on a non-Rolla web interface. It's important to minimize the possibility of mistakes, and not including the full year in the symbol makes things error-prone, and will lead to other options providers winning out.

The 0x [REST interface](#) for swaps has the ability to do a swap by token name rather than by token address. I was unable to figure out whether there was an allow-list of token names, or if it is easy to add a new token. If there is no, or an easily bypassed, access-control for adding new tokens, I would say this finding should be upgraded to high-severity, though I doubt this is the case.

Proof of Concept

```
        /// concatenated symbol string
        tokenSymbol = string(
            abi.encodePacked(
                "ROLLA",
                "-",
                underlying,
                "-",
                _uintToChars(day),
                monthSymbol,
                _uintToChars(year),
                "-",
                displayStrikePrice,
                "-",
                typeSymbol
            )
        );
```

[QTokenStringUtils.sol#L115-L130](#)

```
    /// @return 2 characters that correspond to a number
    function _uintToChars(uint256 _number)
        internal
        pure
        virtual
        returns (string memory)
    {
        if (_number > 99) {
            _number %= 100;
        }

        string memory str = Strings.toString(_number);

        if (_number < 10) {
            return string(abi.encodePacked("0", str));
        }

        return str;
    }
```

[QTokenStringUtils.sol#L181-L199]

Recommended Mitigation Steps

Include the full year in the token's symbol.

[0xca11 (Rolla) confirmed, resolved, and commented]:

> Resolved in [RollaProject/quant-protocol#86].

[M-06] `OperateProxy.callFunction()` should check if the `callee` is a contract

Submitted by WatchPug

[Controller.sol#L550-L558]

```
    /// @notice Allows a sender/signer to make external calls tc
    /// @dev A separate OperateProxy contract is used to make tl
    /// that the Controller, which holds funds and has special p
    /// Protocol, is never the `msg.sender` in any of those exte
    /// @param _callee The address of the contract to be called.
    /// @param _data The calldata to be sent to the contract.
    function _call(address _callee, bytes memory _data) internal
```

```
            IOperateProxy(operateProxy).callFunction(_callee, _data)
        }
```

[OperateProxy.sol#L10-L19]

```
        function callFunction(address callee, bytes memory data) ext
            require(
                callee != address(0),
                "OperateProxy: cannot make function calls to the zer
            );

            (bool success, bytes memory returnData) = address(callee
            require(success, "OperateProxy: low-level call failed");
            emit FunctionCallExecuted(tx.origin, returnData);
        }
```

As the `OperateProxy.sol#callFunction()` function not payable, we believe it's not the desired behavior to call a non-contract address and consider it a successful call.

For example, if a certain business logic requires a successful `token.transferFrom()` call to be made with the `OperateProxy`, if the `token` is not a existing contract, the call will return `success: true` instead of `success: false` and break the caller's assumption and potentially malfunction features or even cause fund loss to users.

The qBridge exploit (January 2022) was caused by a similar issue.

As a reference, OpenZeppelin's `Address.functionCall()` will check and `require(isContract(target), "Address: call to non-contract");`

[https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.5.0/contracts/utils/Address.sol#L135]

```
        function functionCallWithValue(
            address target,
            bytes memory data,
            uint256 value,
            string memory errorMessage
        ) internal returns (bytes memory) {
            require(address(this).balance >= value, "Address: insuff
            require(isContract(target), "Address: call to non-contra

            (bool success, bytes memory returndata) = target.call{va
            return verifyCallResult(success, returndata, errorMessag
        }
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.5.0/contracts/utils/Address.sol#L36-L42

```
        function isContract(address account) internal view returns (
            // This method relies on extcodesize/address.code.length
            // for contracts in construction, since the code is only
            // of the constructor execution.

            return account.code.length > 0;
        }
```

Recommended Mitigation Steps

Consider adding a check and throw when the `callee` is not a contract.

0xca11 (Rolla) confirmed, resolved, and commented:

> Resolved in RollaProject/quant-protocol#85.

[M-07] Low-level transfer via `call()` can fail silently

Submitted by 0xDjango

TimelockController.sol#L414-L415

In the `_call()` function in `TimelockController.sol`, a call is executed with the following code:

```
    function _call(
            bytes32 id,
            uint256 index,
            address target,
            uint256 value,
            bytes memory data
        ) private {
            // solhint-disable-next-line avoid-low-level-calls
            (bool success, ) = target.call{value: value}(data);
            require(success, "TimelockController: underlying transac

            emit CallExecuted(id, index, target, value, data);
        }
```

Per the Solidity docs:

"The low-level functions call, delegatecall and staticcall return true as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed."

Therefore, transfers may fail silently.

Proof of Concept

Please find the documentation here.

Recommended Mitigation Steps

Check for the account's existence prior to transferring.

0xca11 (Rolla) confirmed, resolved, and commented:

> The TimelockController contract was removed from the protocol.

> RollaProject/quant-protocol#90

[M-08] Arbitrary code can be run with Controller as msg.sender

Submitted by hyh

A malicious user can call Controller's operate with ActionType.QTokenPermit, providing a precooked contract address as qToken, that will be called by Controller contract with IQToken(_qToken).permit(), which implementation can be arbitrary as long as IQToken interface and permit signature is implemented.

The Controller is asset bearing contract and it will be msg.sender in this arbitrary permit() function called, which is a setup that better be avoided.

Proof of Concept

When the Controller's operate with a QTokenPermit action, it parses the arguments with Actions library and then calls internal _qTokenPermit:

Controller.sol#L91-L92

_qTokenPermit calls the IQToken(_qToken) address provided without performing any additional checks:

Controller.sol#L497-L516

This way, contrary to the approach used in other actions, qToken isn't checked to be properly created address and is used right away, while the requirement that the address provided should implement IQToken interface and have permit function with a given signature can be easily met with a precooked contract.

Recommended Mitigation Steps

Given that QToken can be called directly please examine the need for QTokenPermit ActionType.

If current approach is based on UI convenience and better be kept, consider probing for IOptionsFactory(optionsFactory).isQToken(_qToken) before calling the address provided.

0xca11 (Rolla) confirmed, resolved, and commented:

> Fixed in RollaProject/quant-protocol#82.

[M-09] Spreads can be minted with a deactivated oracle

Submitted by hyh

When deactivateOracle() is called for an oracle in OracleRegistry it is still available for option spreads minting.

This way a user can continue to mint new options within spreads that rely on an oracle that was deactivated. As economic output of spreads is close to vanilla options, so all users who already posses an option linked to a deactivated oracle can surpass this deactivation, being able to mint new options linked to it as a part of option spreads.

Proof of Concept

Oracle active state is checked with isOracleActive() during option creation in validateOptionParameters() and during option minting in _mintOptionsPosition().

It isn't checked during spreads creation:

FundsCalculator.sol#L91-L117

In other words besides vanilla option minting and creation all spectrum of operations is available for the deactivated oracle assets, including spreads minting, which economically is reasonably close to vanilla minting.

Recommended Mitigation Steps

If oracle deactivation is meant to transfer all related assets to the close only state then consider requiring oracle to be active on spreads minting as well in the same way it's done for vanilla option minting:

Controller.sol#L188-L197

0xca11 (Rolla) confirmed, resolved, and commented:

> Resolved in RollaProject/quant-protocol#81.

[M-10] Admin of the upgradeable proxy contract of `Controller.sol` can rug users

Submitted by WatchPug

Controller.sol#L22-L34

Use of Upgradeable Proxy Contract Structure allows the logic of the contract to be arbitrarily changed.

This allows the proxy admin to perform malicious actions e.g., taking funds from users' wallets up to the allowance limit.

This action can be performed by the malicious/compromised proxy admin without any restriction.

Considering that the purpose of this particular contract is for accounting of the Collateral and LongShortTokens, we believe the users' allowances should not be hold by this upgradeable contract.

Proof of Concept

Given:

- collateral: `USDC`

Rug Users' Allowances

1. Alice `approve()` and `_mintOptionsPosition()` with `1e8 USDC`;
2. Bob `approve()` and `_mintOptionsPosition()` with `5e8 USDC`;
3. A malicious/compromised proxy admin can call `upgradeToAndCall()` on the proxy contract and set a malicious contract as `newImplementation` and stolen all the USDC in Alice and Bob's wallets;

Rug Contract's Holdings (funds that belong to users)

A malicious/compromised proxy admin can just call `upgradeToAndCall()` on the proxy contract and send all the USDC held by the contract to an arbitrary address.

Severity

A smart contract being structured as an upgradeable contract alone is not usually considered as a high severity risk. But given the severe impact (all the funds in the contract and funds in users' wallets can be stolen), we mark it as a `High` severity issue.

Recommended Mitigation Steps

Consider using the non-upgradeable `CollateralToken` contract to hold user's allowances instead.

See also our Recommendation in [issue #49](https://github.com/code-423n4/2022-03-rolla-findings/issues/49).

0xca11 (Rolla) confirmed, resolved, and commented:

> Resolved with the removal of the previous access control role-based system: RollaProject/quant-protocol#90.

alcueca (judge) decreased severity to Medium and commented:

> Assets are not directly at risk, as a governance attack must happen first. Downgraded to medium.

Low Risk and Non-Critical Issues

For this contest, 20 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0x1f8b** received the top score from the judge.

The following wardens also submitted reports: berndartmueller, IllIIIIl, 0xkatana, gzeon, 0xDjango, defsec, danb, badbird, rayn, cryptphi, Ruhum, robee, TerrierLover, WatchPug, 0v3rf10w, hyh, jayjonah8, remora, *and* 0xmint.

[L-01]

It's possible to call the method addAsset multiple times with the same `_underlying` , if you call `addAsset` with empty `symbol` it will bypass the validAsset modifier, and it will be possible to call again the `addAsset` with different values. It will produce a mismatch with the reality and `getAssetsLength`.

This could cause a loss of funds if it is not verified before that this `symbol` is other than empty. It is mandatory to add a require to verify that the `symbol` is not empty. It's also recommended to add a require in _assetSymbol to ensure non-existence tokens are returned.

[L-02]

It's possible to call the method createCollateralToken multiple times with the same `_qTokenAddress`, if you call `createCollateralToken` with empty `_qTokenAddress` it will bypass the require(idToInfo[id].qTokenAddress == address(0)), and it will be possible to call again the `createCollateralToken` with different values. It will produce a mismatch with the reality and `collateralTokenIds`.

It is mandatory to add a require to verify that the `_qTokenAddress` is not empty.

[L-03]

It's possible to approve with metaSetApprovalForAll an empty address for any operator, `ecrecover` is not checked to return `address(0)`, so using `owner=address(0)` it is possible to approve or reject empty owners for any operator. It's recommended to use ECDSA from open-zeppelin.

[L-04]

When calling _getMonth with a value greater than 12 "December" is returned, it's best to make sure the value is as expected.

[L-05]

Use a buggy solidity version with immutables.

The contract uses immutable, and this solidity version defined in the pragma has some issues with them, as you can see here.

[L-06]

It's possible to call the method addAssetOracle multiple times with the same `_asset` , if you call `addAssetOracle` with empty `_oracle` it will bypass the assetOracles[_asset] == address(0), and it will be possible to call again the `addAssetOracle` with different values. It will produce a mismatch with the reality and `assets`

It is mandatory to add a require to verify that the `_oracle` is not empty.

[N-01]

`collateralTokenId` is used as `deadline` and it could be confused, it's better to rename it or add a specific comment about that.

- Actions.sol#L110
- Actions.sol#L135

[N-02]

Use `uint8` for `decimals` in QTokenStringUtils.sol#L142

0xca11 (Rolla) confirmed, resolved, and commented:

> Resolved in RollaProject/quant-protocol@ebe5f2b.

alcueca (judge) commented:

> Score: 100

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top