Tigris Trade Options Audit Report

Prepared by bytes032

Contents

1 About bytes032

bytes032 is an independent smart contract security researcher.

His knack for identifying smart contract security vulnerabilities in a range of protocols is more than a skill; it's a passion. Committed to enhancing the blockchain ecosystem, he invests his time and energy into thorough security research and reviews. Want to know more or collaborate? bytes's always up for a chat about all things security.

Feel free to reach out on X.

2 Protocol Summary

Tigris Trade is a leveraged trading platform that utilizes price data signed by oracles off-chain to provide atomic trades and real-time pair prices. Open positions are minted as NFTs, making them transferable. Tigris is governed by Governance NFT holders. The oracle aggregates real-time spot market prices from CEXs and sign them. Traders include the price data and signature in the trade txs. For people that want to provide liquidity, they can lock up tigAsset tokens (such as tigUSD, received by depositing the appropriate token into the stablevault) for up to 365 days. They will receive trading fees through an allocation of Governance NFTs, which get distributed based on amount locked and lock period.

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

3 Risk Classification

	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

Overview

Project	Tigris Trade Options
Repository	Contracts
Commit	47bf1157f87d...
Date	May 2023

Issues Found

Severity	Count
Critical Risk	3
High Risk	3
Medium Risk	1
Low Risk	2
Informational	0
Gas Optimizations	0
Total Issues	**9**

Summary of Findings

Title	Status
[C-1] Limit order expiry time flaw could lead to loss of funds	Resolved
[C-2] Its possible to open a trade in the same tx with two different prices	Resolved
[C-3] Price validation vulnerability in closeTrade	Resolved
[H-1] OpenInterest can grow infinitely	Resolved
[H-2] A malicious user can open orders exceeding the maximum possible amount	Resolved
[H-3] CloseTrade can deprive users of their rewards	Resolved
[M-1] No check for active Arbitrum Sequencer in getVerifiedPrice	Acknowledged
[L-1] Missing asset existence check	Acknowledged
[L-2] Redundant TradeAsset variables	Acknowledged

5 Findings

5.1 Critical Risk

[C-1] Limit order expiry time flaw could lead to loss of funds

Context: Options.sol

Impact

The vulnerability discovered could lead to significant financial loss for users. In the event that an order is not executed within its specified duration, it can still be executed after the expiry time. This execution could be at an unfavorable price, leading to direct loss of funds.

Description

Right now, a limit order can be initiated with a specific duration, which respectively populates the `expires` variable when the trade is minted.

```
newTrade.duration = _mintTrade.duration;
newTrade.openPrice = _mintTrade.price;
newTrade.expires = _mintTrade.duration + block.timestamp;
```

This variable is crucial, because it's used in `closeTrade` to ensure that the trade can only be closed when it's duration has expired.

```
if (block.timestamp < _trade.expires) revert("!expired");
```

For instance, let's consider the following steps:

1. A user initiates a limit order with a `duration = x`, setting the `expiry = x + block.timestamp`.
2. For some reason, the order cannot be executed for the whole duration, e.g., it enters into one of the `if (trade.openPrice > _price) revert("!limitPrice");` conditions.
3. When the expiry time has passed, the order can now be executed. As soon as it gets executed, it can immediately be closed, because the condition `if (block.timestamp < _trade.expires) revert("!expired");` no longer holds.
4. Depending on the asset's price at the time of execution, this will lead to a direct loss of funds.

Recommendation

To address this vulnerability, it's advised to refactor the function so that `initiateLimitOrder` sets `expires` to 0, irrespective of the duration. Then, `expires` should be updated when the order is executed. This change would prevent orders from being executed after their duration has expired.

Furthermore, it would be beneficial to enforce a minimum duration (e.g., 30 seconds). This would provide a buffer to prevent the immediate execution and closure of trades, allowing users to react to market changes.

[C-2] Its possible to open a trade in the same tx with two different prices

Context: Options.sol

Tigris utilizes oracle nodes which are connected to the Pythnet price feed. The asset prices and other related data is signed and broadcasted directly to Tigris users, which is used to place market orders. This data consists of:

- Asset price
- Spread
- Timestamp
- Node address
- Market open/closed status

- Oracle signature

Upon placing a trade, price data and signature is included in the transaction input parameters, where the validity of the data and the signatures are verified on-chain.

This means there could be two different prices in the "valid signature pool" at the same time.

In `Options.sol`, the price of the asset is fetched through `getVerifiedPrice`

```
        _price = _priceData.price;
        _spread = _priceData.spread;

        if(_withSpreadIsLong == 1 && useSpread)
            _price += _price * _spread / DIVISION_CONSTANT;
        else if(_withSpreadIsLong == 2 && useSpread)
            _price -= _price * _spread / DIVISION_CONSTANT;
```

If a spread is set, it will update the price according to that, e.g. if price is 1000 and spread is 0.1% the long trades will open with price 1001 and short trades will open with price 999.

Otherwise, it will just return the price as is.

When you open a trade, you can specify duration, which is then derived to set the expiration of the trade.

```
>        newTrade.duration = _mintTrade.duration;
        newTrade.openPrice = _mintTrade.price;
>        newTrade.expires = _mintTrade.duration + block.timestamp;
```

Then, when closing a trade the `expiry` variable is used to ensure that only trades which expired can be closed.

```
function closeTrade(
        uint256 _id,
        PriceData calldata _priceData,
        bytes calldata _signature
    )
        external
    {
    ...

>     if (block.timestamp < _trade.expires) revert("!expired");
    ...
```

The catch here is that as per Arbitrum's docs, any timing assumptions a contract makes about block numbers and timestamps should be considered generally reliable in the longer term (i.e., on the order of at least several hours) but unreliable in the shorter term (minutes)

It is unreliable in a shorter term, because if multiple Arbitrum transactions are in a single L1 block, they **will** have the same block.timestamp.

This means around 20 transactions in Arbitrum can have the same block timestamp.

Running

```
cast block --rpc-url https://arb-mainnet.g.alchemy.com/v2/UVXidxBjyLOdMXEJxYtCMqqEkHATR2gQ 17169970
```

Then, running the script for 20 blocks further

```
cast block --rpc-url https://arb-mainnet.g.alchemy.com/v2/UVXidxBjyLOdMXEJxYtCMqqEkHATR2gQ 17169970
```

Yields the following result:

This proves that 20 distinct transactions in Arbitrum can have the same timestamp.

Back to openTrade and closeTrade, this essentially means a user can open/close trade in the same **L1 block**.

Consider the following scenario:

1. useSpread is not set, so the oracle returns the price as is.
2. Amelie opens a trade with where `duration = 0` and `collateral = 1e18` and she picks price X from the pool. Because duration is set to 0, this means expiry = block.timestamp.
3. Immediately, in the same or after the first transaction, she closes her trade, which is possible because `block.timestamp == trade.expiry`, but now picks price Y from the pool, where Y > X.
4. $

Picking a different price from the pool is possible, because anyone can get signed prices. Additionally, the price **timestamp** is not chain dependant, but is generated from the node that is signing the prices.

Hence, the check below will pass for ~20-25 blocks in Arbitrum, because the block.timestamp will be the same, allowing the user to either open/close trades in the same transaction, or do it sequentially, while picking a more favorable price for the closing trade.

```
function getVerifiedPrice(
        uint256 _asset,
        PriceData calldata _priceData,
        bytes calldata _signature,
        uint256 _withSpreadIsLong,
        bool _expirable
    )
        public view
        returns(uint256 _price, uint256 _spread)
    {
        ...
        if(_expirable) require(block.timestamp <= _priceData.timestamp + _validSignatureTimer, "ExpSig");
        ....
```

Recommendation: Update the condition in the `closeTrade` function to prevent a trade from being closed within the same L1 block. You can achieve this by adding a condition to check if the current `block.timestamp` is greater than `_trade.expires`. Here is the suggested change:

```
+     if (block.timestamp <= _trade.expires) revert("!expired");
```

This modification will inhibit the possibility of making profitable transactions by opening and closing a trade within the same block using two different prices from the "valid signature pool".

[C-3] Price validation vulnerability in closeTrade

Context: Options.sol

Impact: This vulnerability could lead to the manipulation of trade outcomes by the users, potentially allowing them to win trades unfairly. As such, it could undermine the integrity of the trading system and expose the Tigris-Trade platform to reputational damage and potential financial losses.

Description: The `closeTrade` function in the Tigris-Trade `Options.sol` contract currently bypasses price expiration checks, enabling users to close trades with potentially stale price data. This is made possible because the `getVerifiedPrice` function call within `closeTrade` has the `_expirable` parameter set to `false`. This means that the timestamp of the price data is not being checked against the current block timestamp.

This could allow users to open a trade at the current price and then close it with a potentially outdated price, essentially gaming the system. In the context of a price expiration condition that should typically prevent this, this can be considered a major security oversight.

The vulnerable code is located here:

```
(uint256 _price,) = getVerifiedPrice(_trade.asset, _priceData, _signature, 0, false);
```

In practice, this means a user can open a trade with whatever the current price is and then select a price that could be days old, but one that will consider the trade as a win.

```
        bool isWin;
        if (_trade.direction && _price > _trade.openPrice) {
            isWin = true;
        } else if(!_trade.direction && _price < _trade.openPrice) {
            isWin = true;
        }
```

Recommendation: To mitigate this vulnerability, it's recommended to activate the price expiration checks in the `closeTrade` function. This could be done by setting the `_expirable` parameter to `true` when calling the `getVerifiedPrice` function in `closeTrade`.

The updated code should be as follows:

~~~
(uint256 _price,) = getVerifiedPrice(_trade.asset, _priceData, _signature, 0, true);



\clearpage
## High Risk


### [H-1] OpenInterest can grow infinitely

**Context:** [Options.sol](https://github.com/Tigris-Trade/Contracts/blob/440009d18ab7c4ac1bdbc87a4381e7e41f380
↪   41a/contracts/options/Options.sol#L48)

**Impact:** This issue could lead to a perpetual increase in openInterest for a specific traded asset without an
↪   appropriate mechanism to decrease it upon trade closure. This flaw can lead to permanent locking of an asset from
↪   trading, disrupting the normal operations of the contract and impacting the platform's liquidity and usability.

**Description**

The `TradedAsset` struct within `Options.sol` is employed to whitelist assets for options trading, setting critical
↪   parameters and constraints for such trades.

```solidity
    struct TradedAsset {
        uint maxCollateral;
        uint minCollateral;
        uint maxDuration;
        uint minDuration;
        uint maxOpen;
        uint openInterest;
        uint assetId;
        uint winPercent;
        uint closeFee;
        uint botFee;
        uint refFee;
    }
~~~

During the execution of a new trade or a limit order, the `openInterest` is properly increased:

```
asset.openInterest += _tradeInfo.collateral;
```

```
        TradedAsset storage asset = tradedAssets[trade.asset];
        asset.openInterest += trade.collateral;
```

This is crucial to ensure that the next open trades/limit orders won't exceed the maximum allowed open orders for that specific asset in both `openTrade` and `executeLimitOrder`

```
function openTrade(
    TradeInfo calldata _tradeInfo,
    PriceData calldata _priceData,
    bytes calldata _signature,
    ERC20PermitData calldata _permitData,
    address _trader
)
    external
{
    _validateProxy(_trader);
    TradedAsset storage asset = tradedAssets[_tradeInfo.asset];

    // asset.openInterest is 0 on creation
    require(asset.openInterest + _tradeInfo.collateral <= asset.maxOpen, "!maxOpen");
    require(_tradeInfo.collateral <= asset.maxCollateral, "!max");
    require(_tradeInfo.collateral >= asset.minCollateral, "!min");
```

```
function initiateLimitOrder(
    TradeInfo calldata _tradeInfo,
    uint256 _orderType, // 1 limit, 2 stop
    uint256 _price,
    ERC20PermitData calldata _permitData,
    address _trader
)
    external
{
    TradedAsset storage asset = tradedAssets[_tradeInfo.asset];
    require(asset.openInterest + _tradeInfo.collateral <= asset.maxOpen, "!maxOpen");
    require(_tradeInfo.collateral <= asset.maxCollateral, "!max");
    require(_tradeInfo.collateral >= asset.minCollateral, "!min");
```

However, the contract doesn't provide a corresponding decrement operation when a trade is closed. This omission allows `openInterest` to grow indefinitely until it reaches the `maxOpen` limit, which blocks any new trades for that asset.

Recommendation: It is recommended to decrement the `openInterest` value appropriately when a trade is closed. The following change should be made to decrease `openInterest` by the amount of the trade's collateral before the token for the position is burned:

```
+       _tradedAsset.openInterest -= _trade.collateral;
    tradeNFT.burn(_id);
    emit TradeClosed(_id, _price, isWin ? _tradedAsset.winPercent : 0, toSend, _trade.trader, _msgSender());
```

[H-2] A malicious user can open orders exceeding the maximum possible amount

Context: Options.sol

Impact: This vulnerability can lead to a significant imbalance in the platform's liquidity management. It allows a malicious user to overcommit resources, potentially destabilizing the system by surpassing the `maxOpen` limit set for an asset.

Description: The described vulnerability lies within the `initiateLimitOrder` function, which initiates limit orders based on the current `openInterest` of the asset. However, the current design does not immediately update the `openInterest` at the time of initiating an order.

```
function initiateLimitOrder(
    TradeInfo calldata _tradeInfo,
    uint256 _orderType, // 1 limit, 2 stop
    uint256 _price,
    ERC20PermitData calldata _permitData,
    address _trader
)
    external
{
    TradedAsset storage asset = tradedAssets[_tradeInfo.asset];
    require(asset.openInterest + _tradeInfo.collateral <= asset.maxOpen, "!maxOpen");
    require(_tradeInfo.collateral <= asset.maxCollateral, "!max");
    require(_tradeInfo.collateral >= asset.minCollateral, "!min");
```

Instead, it is updated at a later stage when executing the order.

```
    TradedAsset storage asset = tradedAssets[trade.asset];
    asset.openInterest += trade.collateral;
```

The problem arises from the fact that when several orders are initiated simultaneously before execution, the check is performed on the same, stale `openInterest` value.

As a result, a user can initiate multiple limit orders which, collectively, would exceed the `maxOpen` limit set for the asset. This situation is not prevented, as each check when initiating an order only considers the `openInterest` at the moment of initiating the order, ignoring subsequent changes due to other orders.

This scenario can be illustrated as follows:

1. Assume `maxOpen = 5e18` and `openInterest = 0`.
2. In a single transaction, Amelie initiates 5 limit orders with 5e18 collateral each.
3. Each `initiateLimitOrder` call passes the check, as it is performed with the stale `openInterest` value.
4. Amelie then executes these 5 transactions, resulting in `openInterest = 25e18`, which significantly exceeds the `maxOpen` limit set by the protocol.

Recommendation: To mitigate this vulnerability, the `openInterest` variable should be updated in real-time as each order is initiated, not at the point of execution. This way, each subsequent `initiateLimitOrder` check would consider the most recent `openInterest` value, including all initiated but not yet executed orders.

Alternatively, if the protocol insists to use the current approach, the following checks should be performed right before execution again:

```
    require(asset.openInterest + _tradeInfo.collateral <= asset.maxOpen, "!maxOpen");
    require(_tradeInfo.collateral <= asset.maxCollateral, "!max");
    require(_tradeInfo.collateral >= asset.minCollateral, "!min");
```

[H-3] CloseTrade can deprive users of their rewards

Context: Options.sol

Impact: The vulnerability affects the determination of winning trades, potentially leading to incorrect outcomes and depriving users of their rewards.

Description: In Tigris's options implementation, users either lose 100% or win 70% (win percent) of their pledged collateral.

The "winning" formula looks like this:

```
function closeTrade(
    uint256 _id,
    PriceData calldata _priceData,
    bytes calldata _signature
)
    external
{
    ...
    bool isWin;
    if (_trade.direction && _price > _trade.openPrice) {
        isWin = true;
    } else if(!_trade.direction && _price < _trade.openPrice) {
        isWin = true;
    }
    ...
```

If trade direction is true, then the option is a *long*, otherwise its a *short*. Dissecting that further, we can observe the price of the trade is a significant factor as well.

So, if a order is a long the price currently fetched by the oracle **must be** higher than the price at the time the option was opened.

On the other hand, if its a **short**, the price **must be** lower than to consider the trade as a win. However, that's not entirely true, as the price actually **must be** lower than OR equal to the price at the time the option was created.

However, currently the function doesn't account of that, meaning it will consider such trades a loss instead of win, thereby depriving users from their rewards.

Recommendation: To address this vulnerability, the code logic needs to be updated to include the equality check. The correct logic for determining a winning trade when the trade direction is short should be as follows:

```
+} else if(!_trade.direction && _price <= _trade.openPrice) {
   isWin = true;
}
```

5.2 Medium Risk

[M-1] No check for active Arbitrum Sequencer in getVerifiedPrice

Context: Options.sol

Impact: The missing Sequencer Uptime Feed check in the `getVerifiedPrice` function potentially exposes users to inaccurate oracle data when the Arbitrum Sequencer is down, making the platform vulnerable to stale pricing attacks.

Description: If the Arbitrum Sequencer goes down, oracle data will not be kept up to date, and thus could become stale. However, users are able to continue to interact with the protocol directly through the L1 optimistic rollup contract. You can review Chainlink docs on L2 Sequencer Uptime Feeds for more details on this.

In the current implementation of the `Options.sol` contract, there is no check for the Sequencer Uptime Feed before the oracle data is returned by the `getVerifiedPrice` function. This could lead to the return of stale data, specifically when the Arbitrum Sequencer goes down.

Under such circumstances, the oracle data would not be updated, and users would continue to interact with the protocol through the L1 optimistic rollup contract.

For instance, if a user who holds tokens worth 1 ETH each initiates a trade, and the sequencer goes down before the trade's expiry blockstamp, the oracle would return stale price data. If the token's price were to drop to 0.5 ETH while the sequencer is down, the bot wouldn't be able to liquidate the user's trade due to the stale price data, thereby exposing the platform to potential losses.

However, given that the chainlink feed can be turned off, this doesn't qualify for more than a medium.

Recommendation: Adapt the following check from Chainlink's documentation:

```
(
            /*uint80 roundID*/,
            int256 answer,
            uint256 startedAt,
            /*uint256 updatedAt*/,
            /*uint80 answeredInRound*/
        ) = sequencerUptimeFeed.latestRoundData();

        // Answer == 0: Sequencer is up
        // Answer == 1: Sequencer is down
        bool isSequencerUp = answer == 0;
        if (!isSequencerUp) {
            revert SequencerDown();
        }

        // Make sure the grace period has passed after the sequencer is back up.
        uint256 timeSinceUp = block.timestamp - startedAt;
        if (timeSinceUp <= GRACE_PERIOD_TIME) {
            revert GracePeriodNotOver();
        }
```

5.3 Low Risk

[L-1] Missing asset existence check

Context: Options.sol

Impact

The `setTradedAsset` function allows adding assets to the system by setting various parameters for the asset. However, it lacks a check to verify whether an asset with the same ID already exists. As a result, if an asset with the same ID is passed to this function, the existing asset will be overwritten with the new values, regardless of whether it was intentional or not.

```
function setTradedAsset(
    uint _id,
    uint _maxC,
    uint _minC,
    uint _maxD,
    uint _minD,
    uint _maxO,
    uint _winP,
    uint[] calldata _fees
) external onlyOwner {
    require(_maxC > _minC, "!C");
    require(_maxD > _minD, "!D");

    TradedAsset storage _asset = tradedAssets[_id];

    _asset.maxCollateral = _maxC;
    _asset.minCollateral = _minC;
    _asset.maxDuration = _maxD;
    _asset.minDuration = _minD;
    _asset.maxOpen = _maxO;
    _asset.assetId = _id;
    _asset.winPercent = _winP;
    _asset.closeFee = _fees[0];
    _asset.botFee = _fees[1];
    _asset.refFee = _fees[2];
}
```

Recommendation

It is recommended to add a check in the `setTradedAsset` function to ensure that an asset with the same ID doesn't already exist in the system. Here is a modified version of the function with the added check:

```
function setTradedAsset(
    uint _id,
    uint _maxC,
    uint _minC,
    uint _maxD,
    uint _minD,
    uint _maxO,
    uint _winP,
    uint[] calldata _fees
) external onlyOwner {
    require(_maxC > _minC, "!C");
    require(_maxD > _minD, "!D");

+   require(tradedAssets[_id].assetId == 0, "Asset already exists");

    TradedAsset storage _asset = tradedAssets[_id];

    _asset.maxCollateral = _maxC;
    _asset.minCollateral = _minC;
    _asset.maxDuration = _maxD;
    _asset.minDuration = _minD;
    _asset.maxOpen = _maxO;
    _asset.assetId = _id;
    _asset.winPercent = _winP;
    _asset.closeFee = _fees[0];
    _asset.botFee = _fees[1];
    _asset.refFee = _fees[2];
}
```

[L-2] Redundant TradeAsset variables

Context: Options.sol

Impact: The current implementation of the `setTradedAsset` function includes the assignment of `maxDuration`, `minDuration`, and `assetId` variables to values, but these variables are unused in the rest of the code. This redundancy may lead to confusion and unnecessary storage consumption.

Description: Within the `setTradedAsset` function, the following lines assign values to the variables `maxDuration`, `minDuration`, and `assetId`:

```
        _asset.maxDuration = _maxD;
        _asset.minDuration = _minD;
        _asset.assetId = _id;
```

However, these variables are not referenced or utilized anywhere else within the function or the surrounding code. As a result, these assignments are redundant and do not serve any purpose in the current implementation.

Recommendation: If that's the intended behavior, it is recommended to remove the unused assignments of `maxDuration`, `minDuration`, and `assetId` from the `setTradedAsset` function.