PUBLIC

Code Assessment

of the Divergence Protocol v1c Smart Contracts

December 15, 2023

Produced for

DIVERGENCE

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Divergence Team,

Thank you for trusting us to help Tenet Technology Ltd with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Divergence Protocol v1c according to Scope to support you in forming an opinion on their security risks.

Tenet Technology Ltd implements an Automated Market Maker(AMM) for digital options. In this system, liquidity providers (LPs) provide liquidity at their positions of interest. Afterwards, traders can buy put or call digital options to take a position whether the price of an underlying asset exceeds the strike price at maturity or not. LPs collect the option premiums and fees paid by traders.

The most critical subjects covered in our audit are functional correctness, solvency of battles, and access control. Security regarding all aforementioned subjects is satisfactory.

The general subjects covered are rounding errors, denial-of-service, documentation and gas efficiency. The security regarding rounding errors is satisfactory, while the security regarding denial-of-service is improvable (see Battles With Malicious Starting Prices). The codebase could be improved regarding gas efficiency (see Gas Optimizations). The documentation and inline code specification can also be improved.

We thank the Tenet Technology Ltd team for always being responsive and very professional during this engagement.

In summary, we find that the codebase provides a satisfactory level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	4
• Code Corrected	2
• Specification Changed	1
• Acknowledged	1
Low-Severity Findings	19
• Code Corrected	10
• Specification Changed	1
• Code Partially Corrected	4
• Risk Accepted	2
• Acknowledged	2

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Divergence Protocol v1c repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	10 Jul 2023	11640136962afb46d35ff19e29502a9dbe48a03e	Initial Version
2	27 Oct 2023	1c0b9317e38bdddb541259d04d0e4a37a6f6ea88	Fixes version
3	15 Dec 2023	f9378a8ba9f4d9d31ba94d08b20be821e16caf4b	Final Version

For the solidity smart contracts, the compiler version `0.8.19` was chosen.

The following files in folders `core` and `periphery` were in the scope:

- core/erros/Errors.sol
- core/libs/*
- core/params/*
- core/token/SToken.sol
- core/types/*
- core/Arena.sol
- core/Battle.sol
- core/Oracle.sol
- core/utils.so;
- periphery/base/*
- periphery/lens/Quoter.sol
- periphery/libs/*
- periphery/params/*
- periphery/types/common.sol
- periphery/Manager.sol

2.1.1 Excluded from scope

Third party libraries (including libraries from Uniswap-V3), tests, and any other files not listed above are excluded from the scope. The new contract `OracleCustom` added in the recent commit is also out of scope. Furthermore, external oracles and any token contract used as collateral were not in the scope of this code assessment. Finally, this code assessment was focused on the correctness of the implementation, however the correctness of the whitepaper and the soundness of the financial model was not evaluated.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Tenet Technology Ltd offers a noncustodial automated market maker (AMM) for options with a predefined payoff structure. To track the relative prices between European digital call and put options, A Uniswap V3 compatible curve is used.

Divergence Protocol v1c's core element is a European digital call or put option. An option gives its holder the right to receive a fixed amount of payout at the maturity. A European digital call option on an underlying asset worth S, with strike price K at maturity T pays **1** collateral token when the price of assets exceeds the strike price at the maturity, else **0** collateral tokens. In contrast, a European digital put option pays **1** token when the price does not reach the predefined strike price, else **0** tokens.

Considering a portfolio holding a European digital call and a European digital put option, it must be valued the same as longing (holding) one collateral, in terms of payout, as only one of the options will have a non-zero payout. A riskless profit can be made if a European digital call and a European digital put can be bought for less than a collateral. Therefore, to remain arbitrage-free, effective prices for European digital puts and calls must add up to one:

$$\frac{\Delta C_0}{\Delta V_0} + \frac{\Delta C_1}{\Delta V_1} = 1$$

Divergence Protocol v1c assumes that, for a given range of the curve, a same amount of call options ΔV_0 (referred as spear) or put options ΔV_1 (referred as shield) can be bought in a transaction ($\Delta V_0 = \Delta V_1$). In this case, the expected amount of payoff for ΔV_0 is the same as ΔV_1:

$$\Delta V_0 = \Delta C_0 + \Delta C_1$$

$$\Delta V_1 = \Delta C_0 + \Delta C_1$$

In this case, ΔC_0 and ΔC_1 represent the collateral corresponding to Token 0 (spear) and Token 1 (shield), respectively. Hence, the price of Token 1 quoted by Token 0 is:

$$P_0^1 = \frac{\Delta C_1}{\Delta C_0}$$

Divergence Protocol v1c makes use of Uniswap v3 virtual curves to represent the collateral quoting Token 0 (ΔC_0) and the collateral quoting Token 1 (ΔV_1). It enables swapping between collateral amounts C_0 and C_1:

$$(C_0 + \frac{L}{\sqrt{P_H}})(C_1 + L\sqrt{P_L}) = L^2$$

By tracking the current `sqrtPrice` $\sqrt{P_0^1}$ and the invariant L, the battle can determine the collateral change for a price change given an amount of liquidity:

$$\Delta C_0 = \Delta\frac{1}{\sqrt{P_0^1}}L, \Delta C_1 = \Delta\sqrt{P_0^1}L$$

These European digital options are implemented using transferrable ERC-20 tokens. Spear is the token representing a European digital call while Shield represents a European digital put. In this text we refer to these ERC-20 tokens as *sToken*, whenever we talk about general behavior of them. Once minted, *sTokens* can be exchanged on a third-party decentralized exchanges (DEXs).

The main actors interacting with Divergence Protocol v1c are:

1. Liquidity Providers (LP). These users deposit their collateral or prior to depositing, buy sTokens and then deposit them into the protocol to provide liquidity in a given range. Any minted European digital option should be backed by the same amount of collateral. LPs take a short position for options that traders buy, in other words, they sell (short) both digital call and put options. Based on the selected price range, they net short either digital calls or puts. When users buy options from a liquidity position owned by an LP, a net owed value gets accrued for the LP. The pool backs any amount minted digital option by the same amount of collateral. LPs can later remove their liquidity before options' maturity. In this case, pool holds a portion of their liquidity to back the shorted options. If these options expire worthless, LPs can reclaim the reserved collateral. Alternatively, LPs can buy sTokens from the pool, and close their net shorts before options' maturity.
2. Traders: Traders pay a premium in collateral to buy (long) a European digital options. If they hold a winning *sToken*, they get the underlying collateral after the battle is settled. In the current implementation, neither a support for trading tokens directly with each-other nor selling tokens back to protocol is supported. But, *sToken* holders can act as an LP and sell them as liquidity to a position. However, since *stokens* are ERC-20 compliant, a third-party DEX can be used for exchanges.

2.2.1 Deployment

Divergence Protocol v1c is organized in two components:

1. Core: The virtual curve is implemented in the core contracts. On high-level, core consists of the following elements:
    1. `Arena`: This contract tracks fee rates, whitelisted collaterals and underlying assets. Also, it serves as a factory to deploy new battles. Fee rates are quoted by `1_000_000` meaning `1_000_000` corresponds to 100%.
    2. `Battle`: Implements the main logic of the virtual curve. This contract is responsible for minting and burning of *sToken*, compute amount in and amount out on trading, collecting fees, and deciding the winning *sToken*. `Battle` instances are deployed through a proxy pattern. Upon deployment, an expiry for each battle is set. These expiries must be at 8 A.M. UTC.
    3. `Oracle`: A smart contract that stores the link between an underlying asset and its respective external oracle. Chainlink oracles are used to query the price of an underlying asset at maturity.
2. Periphery: LPs willing to deposit their collaterals/tokens into the system can interact with the core only through Manager. Hence, user interactions are regulated through the periphery, namely `Manager`.

Considering the functionalities mentioned above, these main contracts should be deployed in the following order:

1. `Oracle` and the contract implementing the logic of `Battle`.
2. `Arena` initialized with `Oracle` address and implementation of `Battle`.
3. `Manager`. After deploying `Manager`, owner of `Arena` must set its address in the contract `Arena`.

Users willing to deploy a new battle can call `createAndInitializeBattle()` with the correct parameters. Maturity times for European digital options must always be set at 8 a.m. UTC.

2.2.2 Adding Liquidity

LPs can add liquidity through `Manager.addLiquidity()`. It goes through the following steps:

1. Finding out liquidity according to the liquidity type (being collateral or sToken), current price, and ticks bordering the position(formulas 20, 21, 22, and 23 in the white paper)

2. Then, it calls into `Battle.mint()`:
    1. Checks that the battle has not expired.
    2. For sToken liquidity, it must hold the correct relation between the current price, price low, and price high.
    3. Updates the position and its bordering ticks. If bordering ticks (t_L and t_H) are not initialized, they would be initialized.
    4. Calls `mintCallback()` of `Manager`, which transfers collateral/sToken held by the LP to the battle.
3. Mint an enumerable NFT for the LP, as a receipt.
4. Add the newly created position to its local mapping `_positions`.

As NFTs are minted with a unique ID, an LP can add liquidity to the same position multiple times which mints new NFTs each time.

2.2.3 Trading

Traders can sell their collateral and receive *sTokens* by calling `Manager.trade()` which performs the following steps:

1. Fetch the corresponding battle defined by its key in `Arena`.
2. Set the caller as `payer`. This address is used later to pull the collateral from and mint *sTokens*.
3. If trader has not specified a price limit, it sets the limit to `TickMath.MIN_SQRT_RATIO + 1` or `TickMath.MAX_SQRT_RATIO - 1` when buying Spear or Shield respectively.
4. Calls into `Battle.trade()`:
    1. Checks that the battle has not expired.
    2. Checks that depending on the direction of trade, price limit does not cross the minimum and maximum ratios.
    3. In an iterative manner, `Battle.trade()` tries to fill the order starting from current tick and keeping the available liquidity into account. Fees and premiums are accounted in the global growth.
    4. Updates `global` and available liquidity.
    5. Calculates amount of collateral (`cAmount`) and sToken (`sAmount`) in the following way:

$$cAmount = amountSpecified - amountSpecifiedRemaining + transactionFee + protocolFee$$

where `amountSpecified` is the collateral amount specified by the user, `amountSpecifiedRemaining` is the unfilled amount of the order. `transactionFee` and `protocolFee` are added to be paid by the trader.

    6. Call `tradeCallback()` on the `Manager` to transfer `cAmount` of collateral from the trader to the `Battle`
    7. Mint the `sAmount` of sToken
5. Performs slippage protection to assure a minimum amount of *sTokens* has been bought.

2.2.4 Removing Liquidity

An LP holding an NFT can call `Manager.removeLiquidity()`. It is necessary the caller of this function be the address holding the NFT. This function is callable if and only if, the state of the position determined by the NFT-ID is `PositionState.LiquidityAdded`. It does the following steps:

1. Calls into `Battle.burn()`.
2. Fetch the remote position in the battle and update the local version through `updateInsideLast`.
3. Calls `getObligation()` to get the amount of collateral removable (formula 33 in white paper)

   $$C_{owed} = C_{seed} + C_{inOwed} - C_{Obligation}$$

   as well as owed amount of sToken (if the initial provided liquidity was of *sToken* type)
4. `Battle.collect()` will finally send the collateral amount to the recipient and mint *sToken*, if any.

By removing the liquidity from a position, its state changes to `PositionState.LiquidityRemoved`.

2.2.5 Redeeming Obligation

Once an LP has removed his liquidity **before** maturity, it might desire to close his net short exposure. Redeeming is only possible as long as the battle is ongoing (not settled). `Manager.redeemObligation()` performs the following actions:

1. Finds the *sToken* having more obligation.
2. Burns the surplus amount of this *sToken* from the caller (buying back the *sToken*).
3. Sends the surplus amount of collateral to the owner of NFT.

Please note that in order to redeem the obligation, any other user holding the surplus amount of the *sToken* can close the net short exposure of the LP. After redeeming obligation, state's position changes to `PositionState.ObligationRedeemed`.

2.2.6 Withdrawing Obligation

After removing liquidity and **after** maturity, LP can unlock the extra amount of the *sToken* which expired out-of-money (losing token); as during removing the liquidity, the maximum obligation of *sTokens* is kept as the obligation. Hence, if more Spears are sold but they expire out-of-money or if more Shields are sold and they expire out-of-money, the difference can be paid back in collateral to the LP, through calling `Manager.withdrawObligation()`. This function changes the state of the position to `PositionState.ObligationWithdrawn`.

An LP can either redeem his obligation before the maturity or withdraw it after maturity. Performing both is mutually exclusive.

2.2.7 Settling the Battle

In order to settle the battle and determine the outcome after maturity, anyone can call `Battle.settle()`. It queries the price of the underlying by calling into `Oracle.getPriceByExternal()`. By comparing the queried price with the preset strike value, outcome of the battle is set.

It is important to note that only the first price after battle's expiration reported by the external oracle is used as the final price to settle the battle.

2.2.8 Exercising

After settling the battle, any user can call `Battle.exercise()` if they hold winning options. *sTokens* are burned and a collateral is paid out to users, deducting a fee if set.

2.2.9 Roles and Trust Model

Owner of Arena: Any account with this role is considered to be fully trusted and always behave in the best interests of the system. `owner` can whitelist collaterals and underlying assets used in battles, update `manager` address in Arena, and collect protocol fees from battles.

Owner of Oracle: This contract is assumed to be trusted and always behave correctly and in the best interests of the protocol. `owner` can set the addresses of external oracles for supported underlying assets and the default prices when external oracles do not work as expected.

We assume roles of `manager` and `arena` in all contracts are held respectively by `Manager.sol` and `Arena.sol`.

Oracles: We assume the external oracles are Chainlink oracles, which are assumed to be trusted and continuously publish correct prices on chain. If oracles misbehave or are malicious, user's funds in battles are at risk.

Collateral assets: They are assumed to be compliant with the ERC20 standard, implement `decimals` function, and be non-malicious. Only ERC20-compliant tokens without special behavior (e.g., transfer callbacks, transfer fees, or inflationary/deflationary tokens) are supported.

Finally, traders and LPs are considered untrusted.

2.3 Changes in Version 2

- When users call `trade()` on the manager side or directly on the battle they have to set a signed integer variable `TradeParams.amountSpecified`. If positive, it means how many *collaterals* (plus some fees added to it at the end) they want to pay and `Battle.trade()` calculates how many *sTokens* they are eligible to receive. And if negative, it dictates how many *sTokens* they want to receive at the end, and then `Battle.trade()` finds out how much *collateral* (plus some fees) they have to pay.
- Oracles have a new functionality, where the owner can manually set prices for a given oracle supporting an underlying at a given timestamp. When settling the battle, the respective Chainlink oracle for the underlying asset is queried. If it fails to report a price between 8 A.M. and 9 A.M. UTC, then these manually set prices are used to resolve the battle result.
- Traders can provide approvals to their trusted accounts to execute actions on behalf of an NFT in Manager.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Traders Pay More Collateral Than Specified Acknowledged

Low-Severity Findings	8

- Unused Imports Code Partially Corrected
- LiquidityDelta Check in Position.update Code Partially Corrected
- Mismatch of Min/Max Ticks With the Edge Prices Acknowledged
- Missing Sanity Checks Code Partially Corrected
- NFTs Are Never Burned in Manager Acknowledged
- Possible to Frontrun Fee Updates in Arena Risk Accepted
- Unclear Specifications for the Format of Strike Price Code Partially Corrected
- Update of Oracle Addresses Risk Accepted

5.1 Traders Pay More Collateral Than Specified

Correctness Medium Version 1 Acknowledged

CS-DPV1-005

Users buying call or put options (spear or shield tokens) specify the amount of collateral they are willing to pay and receive as many options as possible. Traders also pay transaction fees on top of option premiums. Fees are computed as follows:

```
step.feeAmount = FullMath.mulDiv(step.amountOut, fee.transactionFee, 1e6);
...
state.transactionFee += step.feeAmount;
```

Note that `step.amountOut` refers to the amount of options that are bought in a trading step. However, fees are collected in the collateral asset:

```
cAmount = params.amountSpecified - state.amountSpecifiedRemaining
    + state.transactionFee + state.protocolFee;
```

If fees are set for a battle, `cAmount` will be larger than `params.amountSpecified` for a trade, hence the function reverts if traders do not have enough balance (`cAmount`) or they have not provided enough allowance to Manager.

Acknowledged:

Tenet Technology Ltd is aware of this issue but has decided to keep the smart contracts unchanged. Tenet Technology Ltd will inform users in the UI for the fee amount that should be paid on top of the premiums. However, users that interact directly with the smart contracts should be aware of this behavior and take the respective measures.

5.2 Unused Imports

Design | Low | Version 2 | Code Partially Corrected

CS-DPV1-012

Several contracts in the codebase import libraries or contracts that remain unused. We present a non-exhaustive list below:

Manager.sol

```
import { IERC20 } from "@oz/token/ERC20/IERC20.sol";
import { SafeCast } from "@oz/utils/math/SafeCast.sol";
import { IManagerState } from "./interfaces/IManagerState.sol";
```

Arena.sol

```
import { IBattleActions, IBattleMintBurn } from
    "core/interfaces/battle/IBattleActions.sol";
```

Battle.sol

```
import { IBattleBase } from "core/interfaces/battle/IBattleActions.sol";
import { IBattleState } from "core/interfaces/battle/IBattleState.sol";
import { IBattleInit } from "core/interfaces/battle/IBattleInit.sol";
import { IBattleMintBurn } from "core/interfaces/battle/IBattleActions.sol";
import { IArenaState } from "core/interfaces/IArena.sol";
import { DiverSqrtPriceMath } from "core/libs/DiverSqrtPriceMath.sol";
```

Code partially corrected:

Some of the unused imports have been removed, while other are still present. For example:

Manager.sol

```
import { IManagerState } from "./interfaces/IManagerState.sol";
```

Battle.sol

```
import { IBattleBase } from "core/interfaces/battle/IBattleActions.sol";
import { IBattleInit } from "core/interfaces/battle/IBattleInit.sol";
import { IBattleMintBurn } from "core/interfaces/battle/IBattleActions.sol";
```

```
import { IArenaState } from "core/interfaces/IArena.sol";
import { DiverSqrtPriceMath } from "core/libs/DiverSqrtPriceMath.sol";
```

Please note that the aforementioned list is non-exhaustive and there might still exist some other unused imports in the codebase.

5.3 LiquidityDelta Check in `Position.update`

Design | Low | Version 1 | Code Partially Corrected

CS-DPV1-001

The function `Position.update` performs a check whether `liquidityDelta == 0`. Following all callpaths suggests it can never be a case, as `update()` is called from `Battle._updatePosition()` which is called from `Battle._modifyPosition()`. The latest is called only during minting and burning. In `Battle.mint()` an earlier check is done on liquidity being non-zero. Once minted with non-zero liquidity, calling `Manager.removeLiquidity()` assures that the liquidity of this position is also non-zero.

Code partially corrected:

As mentioned above, when minting there already exists a check that the liquidity amount to be added to a position is non-zero, hence minting on the battle-side is also called with a non-zero liquidity. When removing liquidity from a position, as upon adding liquidity, the liquidity is already non-zero. Hence, this check is redundant.

5.4 Mismatch of Min/Max Ticks With the Edge Prices

Correctness | Low | Version 1 | Acknowledged

CS-DPV1-007

The whitepaper suggests that prices `0.01` and `0.99` are valid priced in the curve:

> In implementation P_c^0, P_c^1 are bounded within [0.01, 0.99].

Library TickMath sets the minimum tick to `-45953` and maximum tick to `45953`, which correspond to prices `0.0101` and `0.9899` respectively, hence prices `0.01` and `0.99` cannot be reached in the curve.

Acknowledged:

Tenet Technology Ltd has decided to keep the min/max ticks unchanged, and they provide the following reasoning:

```
The difference in prices computed from min/max ticks and the theoretical limits set
in the whitepaper is so minuscule that it is deemed fit for production.
```

5.5 Missing Sanity Checks

Design | Low | Version 1 | Code Partially Corrected

CS-DPV1-009

The following functions set important state variables but do not implement any sanity check on the inputs:

1. `_fee` in `Arena.setFeeForUnderlying()`.
2. `fee` in `setUnderlyingWhitelist()`.
3. `bk.strikeValue` in `createBattle()`, i.e., after calling `getAdjustPrice()`, `bk.strikeValue` should not be rounded down to 0.
4. `_oracleAddr` and `_battleImpl` in `Arena.constructor()`.
5. `_manager` in `Arena.setManager()`.
6. `_arena` and `_WETH9` in contract `PeripheryImmutableState`.

Code partially corrected:

The sanity checks reported in points 3-6 have been added.

5.6 NFTs Are Never Burned in Manager

Design | Low | Version 1 | Acknowledged

CS-DPV1-010

The NFTs minted in Manager are not burned after an LP removes its liquidity and/or withdraws/redeems its obligations. Hence, the state is not cleared and old NFTs remain in the balance of users, possibly degrading user experience.

Acknowledged:

Tenet Technology Ltd acknowledged this behavior and provided the following reasoning:

> NFTs are kept in order to provide historical records to users, who may need reference for past trade.

5.7 Possible to Frontrun Fee Updates in Arena

Security | Low | Version 1 | Risk Accepted

CS-DPV1-011

The mapping `fees` in the contract `Arena` stores the fees for an underlying. On battle deployment, Arena sets battle's fees for the respective underlying. Fees in the battle cannot be changed after deployment even if they are updated in the Arena. Consider the scenario, in which owner of `Arena` decides to increase fees for an underlying. In this case, users can front-run this transaction to deploy their battles with lower fees, hence making them more attractive for traders.

Risk accepted:

Tenet Technology Ltd is aware of this issue but has decided to keep the relevant codebase unchanged.

5.8 Unclear Specifications for the Format of Strike Price

Correctness | Low | Version 1 | Code Partially Corrected

CS-DPV1-015

Users should specify a strike price when deploying a new battle. The function `createAndInitializeBattle` takes as input a struct which includes the field `battleKey.strikeValue`. This value is expected to be in 18 decimals, however the specifications do not clarify the format of the strike price and its quote token. Incomplete or missing specifications increase the likelihood of mistakes from users or third-party systems that interact with the system.

Code partially corrected:

In Version 2, the following inline comment is added for function `getAdjustPrice`:

```
/// @param price price of underlying. price has decimal 18, eg. eth price 1500 will be 1500 * 10**18
```

However, public and external functions such as `createAndInitializeBattle()` that are called by end users do not describe the expected format of inputs.

5.9 Update of Oracle Addresses

Design | Low | Version 1 | Risk Accepted

CS-DPV1-016

The account `owner` in the contract `Oracle` can update the address of an oracle for an underlying (symbol) via the function `setExternalOracle`. The address change of an external oracle effects only the new battles that are deployed afterwards, while the ongoing battles do not reflect the change. Therefore, ongoing battles cannot settle if the external oracles fail to publish prices as expected. The function `getPriceByExternal` that gets called by battles do not check that `cOracleAddr` is still the correct oracle for the respective symbol.

Rick accepted:

Tenet Technology Ltd is aware of this behavior and has decided to keep the code unchanged, providing the following motivation:

```
The update of oracle addresses during on-going battles is disabled as it is considered
an attack vector, in which a malicious actor may update the oracle address to affect
settlement results. In case of external oracles fail to publish prices as expected,
the owner will be given the ability to address the issue an hour post expiry, only after
all prior processes fail to settle a pool.
```

In Version 2 of the codebase, the severity of the issue presented above is limited as the oracle defaults to prices reported by `owner` in case the external oracle does not work as expected. However, this requires additional trust for `owner` to behave correctly.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	3

- Incorrect Iteration of Rounds From Chainlink Oracle Code Corrected
- Missing Sanity Checks on Price From Oracle Specification Changed
- Missing Slippage Protection When Providing Liquidity Code Corrected

Low-Severity Findings	11

- Events Missing in Arena Code Corrected
- Hardcoded Balances for Spear and Shield in getAllBattles Code Corrected
- Intended Oracle Address for a Battle Code Corrected
- Intended Use of Owed Values in Battle Code Corrected
- Missing Natspec Code Corrected
- Rounding Errors Specification Changed
- Special Behavior of getAdjustPrice Code Corrected
- Special Case in Function Pay Code Corrected
- Storage Variable deploymentParameters in Arena Code Corrected
- Unused Functions in PeripheryPayments Code Corrected
- _safeMint Not Used in Manager Code Corrected

Informational Findings	6

- Indistinguishable Spear and Shield Tokens Code Corrected
- Redundant Import of Library Code Corrected
- Commented Code and Remaining ToDos Code Corrected
- Possible Event Reentrancy in addLiquidity Code Corrected
- Unused Error NotNeedChange Code Corrected
- NFT Approvals Not Considered in Manager Code Corrected

6.1 Incorrect Iteration of Rounds From Chainlink Oracle

Correctness Medium Version 1 Code Corrected

CS-DPV1-002

The internal function `Oracle._getPrice` iterates backward through the rounds reported by a Chainlink oracle as follows:

```
for (uint80 i = id; i > 0; i--) {
    (, int256 answer, uint256 startedAt,,) = cOracle.getRoundData(i);
    ...
}
```

As described in the docs of oracles, there is no guarantee that round IDs are monotonically increasing. More specifically, when the aggregator updates the implementation, a gap in the between two consequent `roundIDs` is created. If such an update happens between a battle's expiry and its settlement, the function `_getPrice` reverts as it queries prices for invalid `roundID`. Therefore, the battle cannot settle and neither liquidity providers, nor traders can claim their collateral tokens.

Code corrected:

In Version 2, the first round in the current phase is fetched and the code iterates backward from the latest ID to the starting ID in the current phase. Hence, all the round IDs are valid. Note, if there is an update of phase ID after the expiry of a battle, oracle returns the manually set prices by its owner.

6.2 Missing Sanity Checks on Price From Oracle

Design | Medium | Version 1 | Specification Changed

CS-DPV1-003

The internal function `Oracle._getPrice` does not enforce a restriction on the maximum delay between the battle expiration and the first price reported by the Chainlink oracle after expiration:

```
for (uint80 i = id; i > 0; i--) {
    (, int256 answer, uint256 startedAt,,) = cOracle.getRoundData(i);
    if (startedAt < ts) {
        break;
    }
    if (startedAt >= ts) {
        p = decimalDiff * answer.toUint256();
        actualTs = startedAt;
    }
}
require(p != 0, "price not exist");
```

It might happen that the Chainlink publishes a price shortly before the expiration and the next overwritten with a significant delay which possibly deviates from the correct price at expiration, however it still gets used as the settlement price by the battle.

Specification changed:

The contract Oracle has been refactored in Version 2 and the specifications of the function `_getPrice` have changed. The new specifications consider prices from the external oracles (i.e., Chainlink) to be valid if they are published in the first hour after the expiration of a battle, otherwise fallback prices provided by the `owner` of the contract Oracle are used.

6.3 Missing Slippage Protection When Providing Liquidity

Security | Medium | Version 1 | Code Corrected

CS-DPV1-004

The function `Manager.addLiquidity` does not implement any slippage protection mechanism, hence leaving LPs susceptible to front-running attacks. In a scenario where an LP intends to provide liquidity into a slot that covers the current price, the transaction can be front-run to move the current price outside the slot such that less `liquidity` is minted for the victim transaction.

Similarly, a victim LP that deploys a new battle with a target initial price and adds liquidity to it can be attacked by front-runners to deploy the same battle (same battle key) but with a malicious initial price such that the honest LP receives less liquidity.

Code corrected:

In Version 2, a slippage protection in the following form has been added to `LiquidityManagement._addLiquidity()`:

```
if (sqrtPriceX96 < params.minSqrtPriceX96 || sqrtPriceX96 > params.maxSqrtPriceX96) {
    revert Errors.Slippage();
}
```

Hence, if due to the front-running, the current price on the battle-side has changed unexpectedly, the transaction reverts.

6.4 Events Missing in Arena

Design | Low | Version 1 | Code Corrected

CS-DPV1-006

The function `setFeeForUnderlying` modifies the state, however the respective event `FeeChanged` declared in the interface `IArena` is not emitted. Similarly, the function `setManager` updates an important state variable but does not emit an event.

Code corrected:

In Arena for evey state changing function, a relevant event gets emitted.

6.5 Hardcoded Balances for Spear and Shield in getAllBattles

Design | Low | Version 1 | Code Corrected

CS-DPV1-036

The function `getAllBattles` queries the state of all battles, except for `spearBalance` and `shieldBalance` which are set to `0`:

```
infos[i] = BattleInfo({
    ...
    spearBalance: 0,
    shieldBalance: 0,
    ...
});
```

However, as the balance of spear and shield in a battle is not defined, holding these values is meaningless.

Code corrected:

Tenet Technology Ltd has removed these fields from the struct `BattleInfo`.

6.6 Intended Oracle Address for a Battle

Design | Low | Version 1 | Code Corrected

CS-DPV1-033

The struct `CreateAndInitBattleParams` includes a field named `oracle`, however the function `createBattle` uses the address in the state variable `oracleAddr` when setting the deployment parameters for a new battle.

Code corrected:

Tenet Technology Ltd has removed the `oracle` field from the struct `CreateAndInitBattleParams`.

6.7 Intended Use of Owed Values in Battle

Design | Low | Version 1 | Code Corrected

CS-DPV1-034

Struct `PositionInfo` has the field `insideLast` that accounts for the growth inside a position per unit of liquidity, and another field `owed` which accounts the growth inside the position. However, the field `owed` is not read by the contracts in scope and appears to be redundant as all positions in battle contracts are held by the manager.

Code corrected:

The unused field `owed` has been removed from the updated codebase.

6.8 Missing Natspec

Design | Low | Version 1 | Code Corrected

CS-DPV1-008

A large number of functions are missing proper documentation and description. Natspec helps to understand more quickly the intention of functions which improves code readability. Natspec of external

functions also helps third-parties that integrate with the system, e.g., by providing information regarding the format of input values.

Code corrected:

The inline comments and specifications have been extended in the Version 3.

6.9 Rounding Errors

Correctness Low Version 1 Specification Changed

CS-DPV1-013

In `Battle._modifyPosition()`, in case of adding collateral liquidity to a position covering the current tick, spear and shield collaterals (`csp` and `csh`) are rounded down. When called from `Battle.mint()`, `seed = csp + csh` is used as the amount of collateral an LP has to pay. Hence, LP pays less collateral than expected.

Specification changed:

The function `_modifyPosition` has been refactored in Version 2 and the computation that was prone to rounding errors has been removed.

6.10 Special Behavior of getAdjustPrice

Correctness Low Version 1 Code Corrected

CS-DPV1-014

The function `getAdjustPrice` behaves differently depending on the value of the input `price`: i) values smaller than `10**11` are rounded down to `0`; ii) values smaller than `10**12` are rounded down to one significant digit; ii) other values are rounded down to two significant digits.

The implementation of `getAdjustPrice()` does not match the formula in the Natspec of the function.

Code corrected:

Tenet Technology Ltd has revised the function to require that input price is larger than `10**12`, so for any allowed price, `getAdjustPrice()` returns the price with 2 decimals.

6.11 Special Case in Function Pay

Design Low Version 1 Code Corrected

CS-DPV1-035

The internal function pay in PeripheryPayments considers a case when `payer == address(this)`.

```
else if (payer == address(this)) {
    // pay with tokens already in the contract (for the exact input multihop case)
    TransferHelper.safeTransfer(tokenAddr, recipient, value);
} else {
    // pull payment
```

```
    TransferHelper.safeTransferFrom(tokenAddr, payer, recipient, value);
}
```

This case is redundant.

Code corrected:

Tenet Technology Ltd has correctly removed this redundant case.

6.12 Storage Variable `deploymentParameters` in Arena

Design | Low | Version 1 | Code Corrected

CS-DPV1-032

The contract `Arena` defines a storage variable named `deploymentParameters`. It is set during `createBattle()` and practically stores the deployment information about the last instance of battle being deployed. This increases gas costs on both deployment and runtime.

Code corrected:

The state variable `deploymentParameters` has been removed from the updated codebase in Version 2.

6.13 Unused Functions in PeripheryPayments

Design | Low | Version 1 | Code Corrected

CS-DPV1-031

The abstract contract PeripheryPayments that is inherited by Manager implements several functions that are declared as `payable`, do not implement any access control, and transfer any token balance held by Manager to arbitrary addresses:

- `unwrapWETH9()`
- `sweepToken()`
- `refundETH`

Code corrected:

The functions listed above have been removed from the updated codebase.

6.14 _safeMint Not Used in Manager

Design | Low | Version 1 | Code Corrected

CS-DPV1-017

The function `Manager.addLiquidity` mints new NFTs by calling the internal function `_mint` which does not perform any check on the `recipient` address. Therefore, if `recipient` is a smart contract that cannot handle NFTs, the newly minted NFT would be locked.

If `recipient` is a smart contract, then the function `_safeMint` checks whether it implements the interface `IERC721Receiver`. Note that `_safeMint` performs a call to untrusted address `recipient`.

Code corrected:

In Version 2, function `_safeMint` is used when creating a new NFT. `_safeMint()` reverts if the recipient of the NFT is a smart contract that does not implement the interface `IERC721Receiver`.

6.15 Possible Event Reentrancy in addLiquidity

Informational | Version 2 | Code Corrected

CS-DPV1-030

The function `addLiquidity` calls `ERC721._safeMint` which triggers the `onERC721Received` hook on the recipient, effectively giving control to an untrusted contract. At this point, the external contract could reenter the contract and call `addLiquidity()` again which mints a new NFT token. In this case, events `LiquidityAdded` would not be ordered sequentially, therefore potentially complicating the monitoring and reconstruction of contract state based on the events info.

Code corrected:

The minting of the NFT has been moved into the end of the function `addLiquidity`.

6.16 Commented Code and Remaining ToDos

Informational | Version 1 | Code Corrected

CS-DPV1-018

Commented out code is present in the function `Battle.trade` and in the contract `Manager`. Also, a `todo` note is remaining in the Natspec of the function `getSTokenDelta`. Removing commented code and addressing remaining note could help improve the readability.

Code corrected:

The commented code and remaining ToDos have been removed from the updated codebase.

6.17 Indistinguishable Spear and Shield Tokens

Informational | Version 1 | Code Corrected

CS-DPV1-022

Arena deploys a set of ERC20 tokens named spear and shield for each battle created. These ERC20 tokens have the same name (`Spear/Shield`) and symbol (`SPEAR/SHIELD`) for all battles, hence possibly confusing for users to distinguish them.

Code corrected:

The function `Arena.createBattle` has been revised to deploy *STokens* with distinguishable name and symbol by appending the battle number after their name and symbol (`Spear-X/Shield-X`):

```
string memory indexString = Strings.toString(battleList.length);
address spear = address(new SToken(string.concat("Spear", indexString),
     string.concat("SPEAR", indexString), decimals, battle));
address shield = address(new SToken(string.concat("Shield", indexString),
     string.concat("SHIELD", indexString), decimals, battle));
```

6.18 NFT Approvals Not Considered in Manager

Informational **Version 1** **Code Corrected**

CS-DPV1-025

The ownership of LP positions in Manager is tracked with NFTs. The contract ERC721 allows holders of NFTs to provide approvals for a specific NFT or all NFTs to other trusted accounts. However, this functionality is not considered in Manager as only the owner of an NFT can call functions that modify the position of an NFT.

Code corrected:

Functions `removeLiquidity()`, `withdrawObligation()` and `redeemObligation` now use the following modifier to restrict the access:

```
modifier isAuthorizedForToken(uint256 tokenId) {
    require(_isApprovedOrOwner(msg.sender, tokenId), "Not approved");
    _;
}
```

Hence, a user holding approvals of NFT, can withdraw the liquidity on behalf of the owner. The recipient of collateral is the onwer of the NFT.

6.19 Redundant Import of Library

Informational **Version 1** **Code Corrected**

CS-DPV1-027

Library `FixedPoint128` is imported twice in the contract Manager.

Code corrected:

The redundant import of this library is removed.

6.20 Unused Error NotNeedChange

Informational **Version 1** **Code Corrected**

CS-DPV1-029

Library Errors declares the error `NotNeedChange` which is unused in the codebase.

Code corrected:

This error has been removed.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Compiler Version

Informational **Version 1**

CS-DPV1-019

The compiler version used (0.8.19) has the following known bugs: https://github.com/ethereum/solidity/blob/develop/docs/bugs_by_version.json#L1834

This is just a note as we do not see any issue applicable to the current code.

The contracts should be deployed using a compiler version they have been thoroughly tested with. Using a very recent version may not be recommend as it may not be considered battle-proof yet. At the time of writing the most recent version is 0.8.21.

For more information please refer to the release notes: https://github.com/ethereum/solidity/releases

7.2 Dependency Versions

Informational **Version 1**

CS-DPV1-020

The smart contract libraries used by the project are:

```
openzeppelin
chainlink
uniswap/v3-core
uniswap/v3-periphery
```

These libraries are either not up-to-date (`openzeppelin` and `chainlink`), or do not refer to a tagged commit (`v3-core` and `v3-periphery`) in the third-party repository.

7.3 Gas Optimizations

Informational **Version 1** **Code Partially Corrected**

CS-DPV1-021

The codebase could be more efficient in terms of gas usage. Reducing the gas costs may improve user experience. Below is an incomplete list of potential gas inefficiencies:

1. The mapping `Oracle.externalOracleOf` is declared `public`, however a public getter function `getCOracle` is redundantly implemented.
2. Function `Oracle.getPriceByExternal` could exit early if `latestRoundData()` returns a price before the option's expiry.
3. Function `Arena.setPermissionless` performs redundant SLOADs when accessing the state variable `isPermissionless`. Instead of toggling the state variable `isPermissionless` it can take the intended value as input parameter.

4. Function `Arena.createBattle` performs redundant SLOADs when accessing the state variable `managerAddr`.
5. Functions `initState` and `init` in the Battle could be merged to avoid one external call from Manager contract to the Battle during deployment.
6. Function `addLiquidity` creates a new memory variable `p1` with the field `recipient` overwritten, however it's not used.
7. The external call `spearAndShield()` in function `_addLiquidity` can be avoided if collateral is being deposited.
8. Function `Battle.mint` performs external calls to verify that the intended amount of tokens has been transferred to battle, however, as the function can be called only by `manager`, `mintCallback` ensures that the funds are transferred.
9. Contract Arena could deploy the pair of `SToken` contracts as proxies to reduce the gas costs of battle deployment. However, note that this approach would increase gas costs on execution.
10. The code assigning the value of `pmMemory.liquidityType` to `bp.liquidityType` in function `removeLiquidity` could be simplified.
11. The modifier `lock` in contract Battle could be more gas efficient if non-zero values are used to record the state.
12. The calculation of `state.protocolFee` could be moved outside the `while-loop`.
13. The check `params.amountRemaining < 0` in function `computeTradeStep` is redundant as `amountRemaining` is of type `uint256`, hence cannot be negative.
14. Function `computeTradeStep` calculates first the capacity (`cap`), then in the `else` branch computes again the same value for `amountIn` (except rounding). The computation of `amountIn` could be avoided if `cap` was rounded up.
15. Function `Manager.tradeCallback` implements a redundant logic with function `verifyCallback`.
16. `Battle.maxLiquidityPerTick` is set to `Tick.tickSpacingToMaxLiquidityPerTick(1)` which has a predefined value. Hence, it can be changed to constant.
17. `Battle._modifyPosition()` calculates the seed using the same formula for Spear and Shield liquidity. Hence, those two can be merge to a single else-statement.
18. `TradeMath.computeTradeStep()` calculates `amountIn` and `amountOut` in the same way, whether `params.amountRemaining < cap` or not. Therefore, those lines calculating `amountIn` and `amountOut` can be moved out of if-else-statement.
19. Input arguments of `Oracle.setExternalOracle()` can be defined as calldata.
20. `sAmount` input argument in `Manager.tradeCallback()` is not used and could be commented out.
21. `data` input argument in `Quoter.tradeCallback()` is not used and could be commented out.
22. `DiverSqrtPriceMath.getSTokenDelta()`, in the current status of the codebase, is always called with non-negative liquidity. Hence, checking for negative liquidity is unnecessary.

Version 2:

23. The field `DeploymentParams.arenaAddr` is not used.
24. Function `Battle.init()` performs two checks to ensure that the battle has not been initialized before.
25. Function `Battle.init()` performs an external call to retrieve decimals of the collateral token which could be passed as function argument from Arena.
26. Function `Battle.mint()` declares a storage pointer `positionInfo` which remains unused.

27. `Arena.createBattle()` checks if `params.bk.strikeValue == 0`. This cannot happen, as `params.bk.strikeValue` is the return value of `getAdjustPrice()` which never returns 0.
28. `Oracle.getPriceByExternal()` can be defined as external.
29. `Position.update()` needs to calculate `liquidityNext` if `liquidityDelta != 0`. Hence, this calculation can be moved inside the if-statement.
30. `Position.tokenId` is a redundant field, as on the manager-side, the respective position is stored in a mapping with the `tokenId` as key.
31. `BattleInitializer.createAndInitializeBattle()` does not need to check if the the battle is already deployed in the Arena, as the same check is done in the Arena.

Code partially corrected:

The codebase has been updated to implement several gas savings reported above. However, points 3, 8, 11, 16, 19-27, 29 and 31 are still present.

7.4 Magic Values in Codebase

Informational **Version 1**

CS-DPV1-023

Several magic values are used in the codebase that could be declared as constant. For instance, the function `Arena.createBattle` uses values `28_800` and `86_400` that could be replaced with constants. Similarly, the contract Battle uses the value `1` for tick spacing which can be replaced with a constant variable to improve code readability.

7.5 Missing Functions in Interfaces

Informational **Version 1**

CS-DPV1-024

The interface `IBattleState` does not include the getter function for the state variable `fee`. Similarly, the interface `IQuoter` does not include the function `quoteExactInput` among others.

7.6 Possible Packing of State Variables

Informational **Version 1**

CS-DPV1-026

Several struct data types in the codebase could be optimized to use less storage by reordering or using smaller types. For instance, the field `expiries` stores timestamps which can be saved in less than 256 bits, hence its type can be changed so that both `collateral` and `expiries` fit into a single storage slot. Similarly, the struct `Fee` could be optimized by changing variable types given that values stored do not exceed `10**6`, while struct `Position` could be optimized by reordering its fields.

7.7 Relaxed Condition on Collateral

Informational **Version 1**

CS-DPV1-037

`Manager.getObligation()` calculates the return value `collateral` in the following way:

```
collateral = pm.owed.collateralIn + pm.seed == obligation ?
    0 : pm.owed.collateralIn + pm.seed - obligation - 1;
```

for liquidity type of collateral. Relaxing the condition by changing == to <= could prevent some corner cases in which due to rounding errors, `pm.owed.collateralIn + pm.seed` is slightly rounded down. The same argument holds for other liquidity types.

7.8 Rounding Errors on Computing sTokens

Informational **Version 1**

CS-DPV1-028

The function `getSTokenDelta` in library DiverSqrtPriceMath takes as last input the flag `roundUp`. The function rounds up all intermediary values when the flag `roundUp` is set to `true`. Therefore, it is possible that the function returns an amount with a higher difference than `1` compared to the amount returned by the same function when called with `roundUp` set to `false`.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Battles Should Be Settled After Expiration

Note Version 1

The contract Battle implements a public function `settle` that can be called by anyone to decide the winning token (spear/shield). However, there is no additional incentive to call this function, hence can be called with a delay.

Function `settle` triggers a call to `Oracle._getPrice()` which implements a `for-loop` that iterates through historical prices reported by the oracle until the correct one (closest to and after the battle's expiration) is found. The loop could run out-of-gas if `settle()` is called with a considerable delay and enough prices have been published by the oracle. In this case, `for-loop` will run out of gas to reach the desired price and will consequently fail.

8.2 Battles With Malicious Starting Prices

Note Version 1

The deployer of a battle can freely choose the initial price `startSqrtPriceX96` as long as it falls in the range of allowed prices. Therefore, it is possible that for an attacker to initialize battles with a malicious price (which does not reflect the fair price of options at the time of deployment) to degrade user experience or cause denial-of-service (DoS). The attacker has to pay for the gas costs to deploy new battles, while the process and cost to recover the battle (set a fair price) depends on the behavior of the attacker.

A battle initialized with a malicious starting price could recover more easily if the attacker did not add liquidity into it. In this scenario, an honest LP should add collateral as liquidity around the fair price and then a trade should happen. However, the trade is limited on buying the option that was priced higher by the attacker on deployment which might be inconvenient for traders.

If an attacker initializes a battle with a malicious price and adds liquidity into it, the process of recovering becomes more costly. In this scenario, the attacker sets the starting price to one edge of the curve (e.g., around minimum tick where spear is very expensive), then adds collateral as liquidity into a range above the current tick where spear is still priced higher than the fair price. The only way for the battle to recover is for traders to consume all liquidity provided by the attacker which means buying options for a price higher than the fair one. Hence, such battles could remain unused, which degrades the user experience.

An attacker can even cause temporary DoS for a pair of collateral and underlying assets. As described in Limited Entropy for Battle Keys, an attacker could deploy all possible battles for a target expiration date (e.g., end of month), and a price change range (±10%) to render such battles useless.

8.3 Fallback Prices in Oracle

Note Version 2

`Oracle.getPriceByExternal()` gets called when the battle is to be settled after the expiry. Tenet Technology Ltd has informed us that they have an off-chain agreement with Chainlink to publish prices between 8 A.M. and 9 A.M. UTC (1 hour interval) for all collateral tokens used in battles. If for any reason

Chainlink fails to publish valid prices during this interval, `Oracle` contract defaults to the `fixPrices` which are manually set by the `owner`.

Furthermore, `getPriceByExternal()` queries Chainlink prices sequencially to find out the first price published after a battle expires. However, if it happens that the aggreggator implementation of the oracle is updated during this time, `getPriceByExternal()` fails to query older prices (due to `phaseId` change). In this scenario also, the contract `Oracle` defaults to the `fixPrices` set by the `owner`.

8.4 Function quoteExactInput Is Gas Inefficient

Note **Version 1**

The function `Quoter.quoteExactInput` is a helper function that allows traders to estimate how much collateral they should pay and how many option tokens they receive for a set of trading parameters. The function is implemented in such a way that it calls the actual `Battle.trade` function to find out `spend` (collateral) and `get` (options) amounts. This means that `quoteExactInput()` triggers all state changes that a normal trade would do, but reverts in the end. Therefore, the gas costs of calling this function on-chain are comparable with gas costs of `Battle.trade`.

8.5 Limited Entropy for Battle Keys

Note **Version 1**

The battle key in Arena is computed as follows:

```
bytes32 battleKeyB32 = keccak256(abi.encode(bk.collateral, bk.underlying, bk.expiries, bk.strikeValue));
```

The entropy for battle keys comes from `expiries` and `strikeValue` for a given pair of `collateral` and `underlying`. However, both `expiries` and `strikeValue` can be values from discrete sets, and it is feasible for one to deploy all possible battles for a target expiry (e.g., end of month) and a price range (e.g., ±10%).

8.6 String Type Used as Key in Mappings

Note **Version 1**

The mappings `externalOracleOf` in `Oracle`, and `fees` and `underlyingWhitelist` in `Arena` use keys of type `string`. We highlight the possibility to have different Unicode characters that render similarly or using invisible Unicode characters, which could be misused by attackers to trick users.

8.7 Users Should Validate Collateral Tokens

Note **Version 2**

If `Arena.isPermissionless` is set, users can deploy battles with arbitrary tokens as collateral. However, the smart contracts in scope of this review work as expected only with standard ERC20 tokens that do not implement special features such as transfer hooks, rebasing, or transfer fees. See Section Roles and Trust Model for more details.