Report

v. 1.0

Customer

Pure.cash

Smart Contract Audit

Pure.cash

3rd October 2024

Report prepared by

ABDK Consulting

Contents

1 Changelog

#	Date	Author	Description
0.1	26.09.24	D. Khovratovich	Initial Draft
0.2	03.10.24	D. Khovratovich	Minor revision
1.0	03.10.24	D. Khovratovich	Release

2 Introduction

The following document provides the result of the audit performed by ABDK Consulting (Mikhail Vladimirov and Dmitry Khovratovich) at the customer request. The audit goal is a general review of the smart contracts structure, critical/major bugs detection and issuing the general recommendations.

Pure.cash is a DeFi protocol that offers a fully decentralized, delta-neutral stablecoin, integrated with innovative long-only perpetual futures, built on Ethereum.

3 Project scope

We were asked to review:

- Original Code
- Code with Fixes

Solidity Files:

libraries/		
LiquidityUtil.sol	PUSDManagerUtil.sol	ReentrancyGuard.sol

4 Methodology

The methodology is not a strict formal procedure, but rather a selection of methods and tactics combined differently and tuned for each particular project, depending on the project structure and technologies used, as well as on client expectations from the audit.

- **General Code Assessment**. The code is reviewed for clarity, consistency, style, and for whether it follows best code practices applicable to the particular programming language used. We check indentation, naming convention, commented code blocks, code duplication, confusing names, confusing, irrelevant, or missing comments etc. At this phase we also understand overall code structure.
- **Entity Usage Analysis**. Usages of various entities defined in the code are analysed. This includes both: internal usages from other parts of the code as well as potential external usages. We check that entities are defined in proper places as well as their visibility scopes and access levels are relevant. At this phase, we understand overall system architecture and how different parts of the code are related to each other.
- **Access Control Analysis**. For those entities, that could be accessed externally, access control measures are analysed. We check that access control is relevant and done properly. At this phase, we understand user roles and permissions, as well as what assets the system ought to protect.
- **Code Logic Analysis**. The code logic of particular functions is analysed for correctness and efficiency. We check if code actually does what it is supposed to do, if that algorithms are optimal and correct, and if proper data types are used. We also make sure that external libraries used in the code are up to date and relevant to the tasks they solve in the code. At this phase we also understand data structures used and the purposes they are used for.

We classify issues by the following severity levels:

- **Critical issue** directly affects the smart contract functionality and may cause a significant loss.
- **Major issue** is either a solid performance problem or a sign of misuse: a slight code modification or environment change may lead to loss of funds or data. Sometimes it is an abuse of unclear code behaviour which should be double checked.
- **Moderate issue** is not an immediate problem, but rather suboptimal performance in edge cases, an obviously bad code practice, or a situation where the code is correct only in certain business flows.
- **Recommendations** contain code style, best practices and other suggestions.

5 Our findings

We found 1 major, and a few less important issues. All identified Major issues have been fixed.

Fixed 1 out of 1 issues

6 Major Issues

CVF-7 FIXED

- **Category** Unclear behavior
- **Source** PUSDManagerUtil.sol

Description It is unclear what validations are meant here. The fact the sizeDelta <= positionCache.size is validated in one of the branches. The fact that payAmount <= positionCache.totalSupply is never explicitly validated.

Recommendation Make validations explicit and put them next to the code that relies on them.

291
```
// never underflow because of the validation above
unchecked {
    position.size = positionCache.size - sizeDelta;
    position.totalSupply = positionCache.totalSupply - payAmount;
}
```

7 Moderate Issues

CVF-1 INFO

- **Category** Flaw
- **Source** LiquidityUtil.sol

Description This value depends on LPToken code, compiler version, and even compiler options. Hardcoding it is very dangerous.

Recommendation Specify the value as: keccak256(type(LPToken).creationCode)

Client Comment *After adjusting to this approach, the contract fails to compile.*

```
18 0xf7ee18f8779e8a47b9fee2bf37816783fe8615833733cf03cc48cd8fc3e3128b;
```

CVF-2 INFO

- **Category** Overflow/Underflow
- **Source** LiquidityUtil.sol

Description Over- and underflow could happen here.

Recommendation Use checked math.

UPDATE: The pnl varaible is int256 and it gets its value from the calcUnrealizedPnL function defined in the PositionUtil library. This function in its documentation comment says: '@return unrealizedPnL The unrealized PnL of the position, positive value means profit, negative value means loss'. Even if the current implementation of the function is never able to return values above certain limit, this isn't guaranteed by the function contract, so this should be considered an implementation detail, and other code shouldn't rely on this fact. Even if the limit would be a part of the function contract, it would be a bad idea to skip compiler-inroduced overflow chaks based on this, as compiler-introduced overflow checks are ments to be used as a second line of defence, that makes it harder to exploit possible bugs in the code. For line 133, relying on caller behavior when writing a function is usually a bad idea, expecialy when this isn't clearly stated in the function contract. This could be an option for low-level performance-optimized functions that clearly states what guarantees they expect form callers. Usually such functions have the word "unsafe" in their names to notify callers that they should use such functions with extra care.

Client Comment *L96: Will not overflow, pnl maximum is type(uint64).max * type(uint128).max*
L98: Will not overflow, netSize maximum is type(uint128).max
L100: Checked at L98
*L130: Will not overflow, pnl maximum is type(uint64).max * type(uint128).max*
L133: Will not overflow, the check has already been performed by the caller

```
96      .mulDiv((pnl + int256(uint256(liquidityBefore))).toUint256(),
           ↪ _param.tokenValue, token.totalSupply())
```

```
98  if (uint256(netSize) + liquidity > liquidityBefore) revert
       ↪ IMarketErrors.BalanceRateCapExceeded();
```

```
100 packedState.lpLiquidity = liquidityBefore - liquidity;
```

```
130     _packedState.lpLiquidity = (int256(uint256(_packedState.
           ↪ lpLiquidity)) + realizedPnL)
```

```
133     _packedState.lpNetSize = netSize - _sizeDelta;
```

CVF-3 INFO

- **Category** Flaw
- **Source** PUSDManagerUtil.sol

Description This value depends on PUSD code, compiler version, and even compiler options. Hardcoding it is very dangerous.

Recommendation Specify the value as: keccak256(type(PUSD).creationCode)

Client Comment *After adjusting to this approach, the contract fails to compile.*

```
21 bytes32 internal constant PUSD_INIT_CODE_HASH =
   0x833a3129a7c49096ba2bc346ab64e2bbec674f4181bf8e6dedfa83aea7fb0fec;
```

CVF-4 INFO

- **Category** Overflow/Underflow
- **Source** PUSDManagerUtil.sol

Description Overflow is possible here.

Recommendation Use checked math.

UPDATE: It is not clear where it is guaranteed that tradingFeeRate doesn't exceed BASIS_POINTS_DIVISOR. Probably this check is performed in another file or even outside the audit scope. Anyway, relying on such busness-level contraints when skipping compiler-introduced overflow checks is a bad idea. See comment for CVF-2 for explanation. For line 90, the value for spreadX96 is retruned by the refreshSpread function that doesn't guarantee any uppoer limit by its contract, so even if such limit actually exists, it should be considered an implementaion detail.

Client Comment *L89: Will not overflow, tradingFeeRate has already been checked during setup to ensure it won't exceed BASIS_POINTS_DIVISOR*
L90: Will not overflow, spreadX96 maximum is type(uint128).max « 96
L118: Will not overflow, minMintingRate has already been checked during setup to ensure it won't exceed BASIS_POINTS_DIVISOR
*L264: Will not overflow, pnl maximum is type(uint64).max*type(uint96).max*
*L347: Will not overflow, pnl maximum is type(uint64).max*type(uint96).max*
*L475: Will not overflow, numeratorX96 maximum is 1e7*2^96*
L492: Will not overflow, maxBurningRatehas already been checked during setup to ensure it won't exceed BASIS_POINTS_DIVISOR
L516: Will not overflow

```
89      uint256 denominatorX96 = (uint256(Constants.BASIS_POINTS_DIVISOR
           ↪ ) + _cfg.tradingFeeRate) << 96;
90      denominatorX96 += spreadX96 * Constants.BASIS_POINTS_DIVISOR;
```

```
118     (uint256(_cfg.minMintingRate) * lpLiquidity) / Constants.
           ↪ BASIS_POINTS_DIVISOR
```

```
264     int256 receiveAmountInt = int256(uint256(sizeDelta)) +
           ↪ realizedPnL;
```

```
347 int256 receiveAmountInt = int256(uint256(_param.sizeDelta)) - int256
       ↪ (uint256(tradingFee)) + realizedPnL;
```

```
475 numeratorX96 *= _param.entryPrice;
```

```
492 uint256 maxBurningSizeCap = (uint256(lpLiquidity) * _maxBurningRate)
       ↪  / Constants.BASIS_POINTS_DIVISOR;
```

```
516 uint256 totalSupplyAfter_ = uint256(_totalSupply) + _amountDelta;
```

CVF-5 INFO

- **Category** Suboptimal
- **Source** PUSDManagerUtil.sol

Description Here a business-level constraint prevents low-level overflow, which is a bad practice, as it introduces hidden relationship between code parts.

Recommendation Use checked math.

Client Comment *For gas savings.*
Recommand downgrade.

```
168 // Because the short position is always less than or equal to the
        ↪ long position,
    // there will be no overflow
170 position.size = sizeBefore + sizeDelta;
```

CVF-6 INFO

- **Category** Overflow/Underflow
- **Source** PUSDManagerUtil.sol

Description Overflow is possible here.

Recommendation Use safe conversion.

Client Comment *L168: Will not overflow, checked at L211*
L351: Will not overflow, calculated based on proportion
L404: Will not overflow, calculated based on proportion

```
217 payAmount = uint64(_param.amount);
```

```
351 pusdDebtDelta = uint64(
        Math.ceilDiv(uint256(_param.sizeDelta) * positionCache.
            ↪ totalSupply, positionCache.size)
    );
```

```
404 receiveAmount = uint128((uint256(moduleCache.tokenPayback) * amount)
        ↪  / moduleCache.pusdDebt);
```

CVF-8 INFO

- **Category** Overflow/Underflow
- **Source** PUSDManagerUtil.sol

Description Underflow is possible here.

Recommendation Use checked math.

Client Comment *L355: Will not underflow, the check has already been performed by the caller*
L356: Will not underflow, pusdDebtDelta is calculated based on proportion
L405: Will not underflow, receiveAmount is calculated based on proportion
L474: Will not underflow, tradingFeeRate has already been checked during setup to ensure it won't exceed BASIS_POINTS_DIVISOR

```
355 position.size = positionCache.size - _param.sizeDelta;
    position.totalSupply = positionCache.totalSupply - pusdDebtDelta;

405 module.tokenPayback = moduleCache.tokenPayback - receiveAmount;

474 uint256 numeratorX96 = uint256(Constants.BASIS_POINTS_DIVISOR -
      ↪ _param.tradingFeeRate) << 96;
```

CVF-9 INFO

- **Category** Procedural
- **Source** LiquidityUtil.sol

Description The "IConfigurable" interface isn't the audit scope, so we don't know the types of MarketConfig fields, which, in turn, doesn't allow us to ensure safety of unchecked blocks.

Client Comment *The file only contains some very simple configurations and does not require auditing.*

```
51 IConfigurable.MarketConfig storage _cfg,
```

CVF-10 INFO

- **Category** Procedural
- **Source** PUSDManagerUtil.sol

Description The "IConfigurable" interface isn't the audit scope, so we don't know the types of MarketConfig fields, which, in turn, doesn't allow us to ensure safety of unchecked blocks.

Client Comment *The file only contains some very simple configurations and does not require auditing.*

```
70  IConfigurable.MarketConfig storage _cfg,
```

```
192 IConfigurable.MarketConfig storage _cfg,
```

CVF-11 INFO

- **Category** Overflow/Underflow
- **Source** PUSDManagerUtil.sol

Description Phantom overflow is possible here, i.e. a situation when the final calculation result would fit into the destination type, while some intermediary calculation overflows.

Recommendation Use the "mulDiv" function.

Client Comment *L215: Will not overflow, maximum is type(uint96).max * type(uint128).max*
*L404: Will not overflow, maximum is type(uint128).max * type(uint128).max*
*L492: Will not overflow, maximum is type(uint128).max * 1e7*

```
215     sizeDelta = ((uint256(_param.amount) * positionCache.size) /
          ↪ positionCache.totalSupply).toUint96();
```

```
404 receiveAmount = uint128((uint256(moduleCache.tokenPayback) * amount)
      ↪  / moduleCache.pusdDebt);
```

```
492 uint256 maxBurningSizeCap = (uint256(lpLiquidity) * _maxBurningRate)
      ↪  / Constants.BASIS_POINTS_DIVISOR;
```

8 Minor Issues

CVF-12 INFO

- **Category** Bad datatype
- **Source** LiquidityUtil.sol

Recommendation The return type should be “ILPToken”.

Client Comment *Ignored*

```
36 function deployLPToken(IERC20 _market, string calldata _tokenSymbol)
   ↪  public returns (LPToken token) {
```

```
41 function computeLPTokenAddress(IERC20 _market) internal view returns
   ↪  (address) {
```

```
45 function computeLPTokenAddress(IERC20 _market, address _deployer)
   ↪ internal pure returns (address) {
```

CVF-13 INFO

- **Category** Bad datatype
- **Source** PUSDManagerUtil.sol

Recommendation The return type should be “IPUSD”.

Client Comment *Ignored*

```
56 function deployPUSD() public returns (PUSD pusd) {
```

```
60 function computePUSDAddress() internal view returns (address) {
```

```
64 function computePUSDAddress(address _deployer) internal pure returns
   ↪  (address) {
```

CVF-14 INFO

- **Category** Procedural
- **Source** PUSDManagerUtil.sol

Recommendation Brackets around multiplication are redundant.

Client Comment *These brackets were added by Prettier.*

```
215 sizeDelta = ((uint256(_param.amount) * positionCache.size) /
    ↪ positionCache.totalSupply).toUint96();
```

CVF-15 INFO

- **Category** Suboptimal
- **Source** PUSDManagerUtil.sol

Description In ERC20 the “decimals” property is used by UI to render token amounts in a human readable way. Using this property in smart contracts is discouraged.

Recommendation Treat all token amounts as integers.

Client Comment *Decimals is needed when exchanging tokens.*

```
418 uint8 decimals = IERC20Metadata(address(_collateral)).decimals();
```

CVF-16 INFO

- **Category** Procedural
- **Source** ReentrancyGuard.sol

Description This version requirement is inconsistent with other files.

Recommendation Use the consistent version requirements across code base.

Client Comment *The transient store is only supported starting from this version.*

```
2 pragma solidity ^0.8.24;
```

CVF-17 INFO

- **Category** Procedural
- **Source** ReentrancyGuard.sol

Description Consider specifying as “^0.8.0” unless there is something special regarding this particular version.

Client Comment *The repeated issue.*

```
2 pragma solidity ^0.8.24;
```

ABDK
Consulting

About us

Established in 2016, is a leading service provider in the space of blockchain development and audit. It has contributed to numerous blockchain projects, and co-authored some widely known blockchain primitives like Poseidon hash function.

The ABDK Audit Team, led by Mikhail Vladimirov and Dmitry Khovratovich, has conducted over 40 audits of blockchain projects in Solidity, Rust, Circom, C++, JavaScript, and other languages.

Contact

Email

dmitry@abdkconsulting.com

Website

abdk.consulting

Twitter

twitter.com/ABDKconsulting

LinkedIn

linkedin.com/company/abdk-consulting