Shell Protocol
Findings & Analysis Report

2023-10-04

Table of contents

Overview

About C4

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

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

During the audit outlined in this document, C4 conducted an analysis of the Shell Protocol smart contract system written in Solidity. The audit took place between August 21 — August 28 2023.

Wardens

44 Wardens contributed reports to the Shell Protocol:

1. pontifex
2. Mirror
3. Udsen
4. oakcobalt
5. prapandey031
6. Testerbot
7. d3e4
8. ItsNio
9. ktg
10. markus_ether
11. mert_eren
12. T1MOH
13. skodi
14. 0xSmartContract
15. lsaudit
16. MrPotatoMagic

17. catellatech
18. ihtishamsudo
19. MohammedRizwan
20. pfapostol
21. plainshift (thank_you, windhustler and surya)
22. 0xmystery
23. JP_Courses
24. carrotsmuggler
25. rjs
26. moneyversed
27. 0xAnah
28. 0xhacksmithh
29. 0x11singh99
30. 0x4non
31. Sathish9098
32. Jorgect
33. epistkr
34. 0xprinc
35. Fulum
36. lanrebayode77
37. Rolezn
38. Shubham
39. ast3ros
40. nisedo
41. MatricksDeCoder
42. chainsnake

This audit was judged by Dravee.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 1 unique vulnerability, receiving a risk rating in the category of HIGH severity.

Additionally, C4 analysis included 21 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 13 reports recommending gas optimizations.

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

Scope

The code under review can be found within the C4 Shell Protocol repository, and is composed of 1 smart contract written in the Solidity programming language and includes 460 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

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

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (1)

[H-01] Lack of Balance Validation

Submitted by Mirror, *also found by* prapandey031, d3e4, Udsen, ItsNio, *pontifex (*1, 2*),* ktg, markus_ether, Testerbot, mert_eren, T1MOH, oakcobalt, *and* skodi

Description

The pool's ratio of y to x must be within the interval `[MIN_M, MAX_M)`, which will be checked by the `_checkBalances()` function. External view functions will call the `_swap()`, `_reserveTokenSpecified()` or `_lpTokenSpecified()` functions to get the specified result. However, `_checkBalances()` is only used in the `_swap()` and `_lpTokenSpecified()` functions. There is no balance validation for `depositGivenInputAmount()` and `withdrawGivenOutputAmount()` functions, which use the `_reserveTokenSpecified()` function.

Impact

If are no other validations outside of these two functions, user deposits/withdrawls may break the invariant, i.e. the pool's ratio of y to x is outside the interval `[MIN_M, MAX_M)`.

Proof of Concept

Add the following code in the test/EvolvingProteusProperties.t.sol file to the EvolvingProteusProperties contract, and run `forge test --mt RatioOutsideExpectedInterval`:

```
    function testDepositRatioOutsideExpectedInterval(uint256 x0, uir
      int128 MIN_M = 0x00000000000002af31dc461;
      uint256 INT_MAX_SQRT = 0xb504f333f9de6484597d89b3754abe9f;

      vm.assume(x0 >= MIN_BALANCE && x0 <= INT_MAX_SQRT);
      vm.assume(y0 >= MIN_BALANCE && y0 <= INT_MAX_SQRT);
      vm.assume(s0 >= MIN_BALANCE && s0 <= INT_MAX_SQRT);
      vm.assume(depositedAmount >= MIN_OPERATING_AMOUNT && deposited
      vm.assume(y0/x0 <= MAX_BALANCE_AMOUNT_RATIO);
      vm.assume(x0/y0 <= MAX_BALANCE_AMOUNT_RATIO);
      vm.assume(int256(y0).divi(int256(x0) + int256(depositedAmount)
      SpecifiedToken depositedToken = SpecifiedToken.X;

      vm.expectRevert();  // There should be at least one case that
      DUT.depositGivenInputAmount(
```

```
        x0,
        y0,
        s0,
        depositedAmount,
        depositedToken
    );
  }

  function testWithdrawRatioOutsideExpectedInterval(uint256 x0, ui
    int128 MIN_M = 0x00000000000002af31dc461;
    uint256 INT_MAX_SQRT = 0xb504f333f9de6484597d89b3754abe9f;

    vm.assume(x0 >= MIN_BALANCE && x0 <= INT_MAX_SQRT);
    vm.assume(y0 >= MIN_BALANCE && y0 <= INT_MAX_SQRT);
    vm.assume(s0 >= MIN_BALANCE && s0 <= INT_MAX_SQRT);
    vm.assume(withdrawnAmount >= MIN_OPERATING_AMOUNT && withdrawr
    vm.assume(y0/x0 <= MAX_BALANCE_AMOUNT_RATIO);
    vm.assume(x0/y0 <= MAX_BALANCE_AMOUNT_RATIO);
    vm.assume(withdrawnAmount < y0);    // no more than balance
    vm.assume((int256(y0) - int256(withdrawnAmount)).divi(int256(x
    SpecifiedToken withdrawnToken = SpecifiedToken.Y;

    vm.expectRevert();  // There should be at least one case that
    DUT.withdrawGivenOutputAmount(
        x0,
        y0,
        s0,
        withdrawnAmount,
        withdrawnToken
    );
  }
```

Recommended Mitigation Steps

It's recommended to add `_checkBalances(xi + specifiedAmount, yi)` after L579 and add `_checkBalances(xi, yi + specifiedAmount)` after L582.

Assessed type

Invalid Validation

viraj124 (Shell) commented via duplicate Issue #268:

> This should be low/med at best IMO. We're adding the balance check, but note `_getUtility` is an internal method and there are input checks of the reserve balances values passed, so this is an invalid argument.

viraj124 (Shell) confirmed and commented via duplicate Issue #268:

> We checked this further with some other members of the team and agreed this is high severity. We've fixed this in a PR.

Dravee (judge) increased severity to High

Low Risk and Non-Critical Issues

For this audit, 21 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Udsen** received the top score from the judge.

The following wardens also submitted reports: prapandey031, MrPotatoMagic, 0xSmartContract, lsaudit, MohammedRizwan, JP_Courses, 0xprinc, Fulum, plainshift, pontifex, Testerbot, lanrebayode77, Rolezn, Shubham, ast3ros, 0xmystery, nisedo, Mirror, MatricksDeCoder, *and* chainsnake.

[01] RECOMMENDED TO ADD AN INPUT VALIDATION FOR THE `duration` VARIABLE IN THE `constructor`

In the `EvolvingProteus.constructor` the `duration` is passed in as an input parameter. Function `duration` denotes the total duration of the curve's evolution. Hence, the `duration` variable is a critical variable of the protocol. If duration is set to zero by mistake, it will prompt a redeployment of the contract which could be waste of gas and additional workload. An input validity check should be performed for the `duration` variable inside the `constructor` as follows:

```
require(duration != 0, `Duration can not be zero`);
```

https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L243-L263

Note from the judge: for a more detailed description of this finding, please see issue 118 from warden lsaudit

[02] DISCREPANCY BETWEEN NATSPEC COMMENTS AND LOGIC IMPLEMENTATION

The `EvolvingProteus._getUtility` function is used to calculate the `utility` value of the pool at the time of the function call. The `utility` is calculated using a quadratic formula which is shown below:

```
k(ab - 1)u**2 + (ay + bx)u + xy/k = 0
```

As per the equation, there is a `k` value used in the `a` and `c` coefficients. But this `k` value is not used when calculating the `aQuad` and the `cQuad` values. This could be due to `normalization` of the equation or the `k = 1`. But the reason for the missing `k` value from the coefficient calculations is not given.

Hence, it is fair to assume there is a discrepancy between the Natspec comments and the actual implementation of the logic.

https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L697
https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L712-L714

[03] DISCREPANCY BETWEEN NATSPEC COMMENT AND LOGIC IMPLEMENTATION OF THE `withdrawGivenInputAmount` FUNCTION

The `natspec` comments of the `EvolvingProteus.withdrawGivenInputAmount` function states that the `feeDirection` of the function to be used is `FEE_UP`. The reason for using `FEE_UP` is described as wanting to increase the perceived amount of reserve tokens leaving the pool; however, the `feeDirection` is determined by the protocol to benefit itself. The `withdrawGivenInputAmount` transaction will benefit the protocol if the perceived amount of reserve tokens leaving the pool is decreased, which is equivalent to charging a fee from the withdrawing amount.

The logic implementation of the `withdrawGivenInputAmount` function has implemented this requirement correctly, as shown below:

```
int256 result = _lpTokenSpecified( //@audit-info - amount of
    withdrawnToken,
    -int256(burnedAmount),
    FEE_DOWN, //@audit-issue - discrepancy between the NATSP
    int256(totalSupply),
    int256(xBalance),
    int256(yBalance)
);
```

It is recommended to update the `natspect` comments of the `withdrawGivenInputAmount` function to state that the `feeDirection` used for the `withdrawGivenInputAmount` transaction is `FEE_DOWN`, by providing it with the proper reason to do so.

https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L459-L461
https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L478-L485

[04] ERRONEOUS NATSPEC COMMENTS SHOULD BE CORRECTED

```
@param px_final The final price at the `y axis`
```

Above comment should be corrected as follows:

```
@param px_final The final price at the `x axis`
```

https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L240

```
    The minimum price value calculated with abdk library equivalent
```

The above comment should be corrected as follows, since the given value (`10^12`) in the comment is not the minimum price value:

```
    The minimum price value calculated with abdk library equivalent
```

https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L173

```
     *  without caring about which particular function `is was` call
```

The above comment should be corrected as follows:

```
     *  without caring about which particular function `was` called.
```

https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L736
https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L768

[05] ADD MEANINGFUL `revert` MESSAGES TO THE `require` STATEMENTS

In the `EvolvingProteus` contract, there are multiple occasions where `require` statements are used for conditional checks. But these `require` statements are not using `revert messages` . Hence, if these `require` statements revert, it will be difficult to find the `reason for the revert` .

It is recommended to add meaningful revert messages to these `require` statements.

https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L414
https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L397-L402

[06] `MIN_BALANCE` INVARIANT IS NOT CHECKED FOR THE FINAL LP TOKEN SUPPLY AMOUNT IN THE `_reserveTokenSpecified` FUNCTION, THUS BREAKING EXPECTED BEHAVIOUR OF THE PROTOCOL

The `EvolvingProteus._reserveTokenSpecified` function is called to calculate the `changed amount` of the `LP token supply` based on the `proportional change of the utility` of the pool.

Firstly the initial utility and final utility are calculated as shown below:

```
ui = _getUtility(xi, yi);
uf = _getUtility(xf, yf);
```

Then, the proportional change of the utility is used to calculate the final LP token supply amount as shown below:

```
uint256 result = Math.mulDiv(uint256(uf), uint256(si), uint2
require(result < INT_MAX);
int256 sf = int256(result);
```

The issue here is the computed `sf` (final LP token supply amount) is not checked against the `MIN_BALANCE` value. Since `_reserveTokenSpecified` is called by both `EvolvingProteus.depositGivenInputAmount` and `EvolvingProteus.withdrawGivenOutputAmount` functions, the `sf` amount can decrease from the initial LP token supply amount of `si` during reserve token withdrawals.

This could prompt the `sf < MIN_BALANCE` condition to occur. If this condition occurs, the transaction should revert as it is done in the

`EvolvingProteus._getUtilityFinalLp` function. But the transaction will not revert in the `_reserveTokenSpecified` function execution since there is no conditional check to verify `sf >= MIN_BALANCE`.

The `withdrawGivenOutputAmount` transaction will execute successfully without reverting, even if the key invariant `sf >= MIN_BALANCE` is broken. Hence, this breaks the expected behaviour of the protocol.

Proof of Concept

```
ui = _getUtility(xi, yi);
uf = _getUtility(xf, yf);

uint256 result = Math.mulDiv(uint256(uf), uint256(si), u
require(result < INT_MAX);
int256 sf = int256(result);

// apply fee to the computed amount
computedAmount = _applyFeeByRounding(sf - si, feeDirecti
```

https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L586-L594

```
require(sf >= MIN_BALANCE);
```

https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L658

```
int256 result = _reserveTokenSpecified(
    withdrawnToken,
    -int256(withdrawnAmount),
    FEE_UP,
    int256(totalSupply),
    int256(xBalance),
    int256(yBalance)
);
```

https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L441-L448

Tools Used

VSCode

Recommended Mitigation Steps

It is recommended to add the conditional check to verify `sf >= MIN_BALANCE` in the `_reserveTokenSpecified` function, after the `sf` value is calculated, as shown below. If the invariant for `MIN_BALANCE` is broken, then the transaction should revert.

```
uint256 result = Math.mulDiv(uint256(uf), uint256(si), uint2
require(result < INT_MAX);
int256 sf = int256(result);

require(sf >= MIN_BALANCE);
```

[07] `_checkBalances` FUNCTION BREAKS THE EXPECTED BEHAVIOUR OF THE PROTOCOL DUE TO ERRONEOUS CONDITIONAL CHECK

The `EvolvingProteus._checkBalances` function is used to verify the token reserve balances and the token reserve ratio are within the valid boundaries of the pool. To validate the `reserve ratio` is within the valid range of `MIN_M - MAX_M`, the following checks are performed:

```
if (finalBalanceRatio < MIN_M) revert BoundaryError(x,y);
else if (MAX_M <= finalBalanceRatio) revert BoundaryError(x,
```

The `NATSPEC` comments for the `MIN_M` and `MAX_M` are given as follows:

```
MIN_M -> This limits the pool to having at most 10**8 x for each
MAX_M -> This limits the pool to having at most 10**8 y for each
```

Even though the documentation says the `y / x` ratio can have at most `10**8`, the logic implementation reverts when the `y / x == 10**8` as shown below:

```
else if (MAX_M <= finalBalanceRatio) revert BoundaryError(x,
```

But it is not the case with the `MIN_M`, as the transaction will not revert when `y / x == 1 / 10**8`. There is a discrepancy between the documentation and the logic implementation.

Consider a scenario where the `y balance = 200 * 10**20` and `x balance = 200 * 10**12`; now `y / x = 10**8`. The transaction is not expected to be reverted since the documentation mentions that the pool is allowed to have at most `10**8` y for each x. But according to the logic implementation, this transaction will revert since `MAX_M == finalBalanceRatio`. This breaks the expected behaviour of the protocol.

Proof of Concept

```
if (finalBalanceRatio < MIN_M) revert BoundaryError(x,y)
else if (MAX_M <= finalBalanceRatio) revert BoundaryErrc
```

https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L812-L813

```
 This limits the pool to having at most 10**8 y for each x.
*/
int128 constant MAX_M = 0x5f5e100000000000000000;
```

https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L155-L157

```
 When a token has 18 decimals, this is one microtoken
*/
```

```
int256 constant MIN_BALANCE = 10**12;
```

https://github.com/code-423n4/2023-08-shell/blob/main/src/proteus/EvolvingProteus.sol#L149-L151

Tools Used

VSCode

Recommended Mitigation Steps

It is recommended to modify the `MAX_M <= finalBalanceRatio` conditional check of the `_checkBalances` function, as shown below (omit the equality check in it):

```
else if (MAX_M < finalBalanceRatio) revert BoundaryError(x,
```

Dravee (judge) commented:

> 01 - Low (0-input validation).
> 02 - Low (Issue with comment).
> 03 - Low (Issue with comment).
> 04 - 3 Lows (Issue with comment).
> 05 - Out of scope - due to the Automated Findings Report.
> 06 - Low
> 07 - Info (accepting as typo in comment but with a decreased severity as the warden misunderstood).

Gas Optimizations

For this audit, 13 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **lsaudit** received the top score from the judge.

The following wardens also submitted reports: MrPotatoMagic, JP_Courses, 0xAnah, 0xhacksmithh, 0x11singh99, pontifex, 0x4non, pfapostol, 0xSmartContract, Sathish9098, Jorgect, *and* epistkr.

[G-01] Do not calculate constants

```
146: uint256 constant INT_MAX = uint256(type(int256).max);
151: int256 constant MIN_BALANCE = 10**12;
181: uint256 constant MAX_BALANCE_AMOUNT_RATIO = 10**11;
191: uint256 constant FIXED_FEE = 10**9;
196: int256 constant MULTIPLIER = 1e18;
201: int256 constant MAX_PRICE_RATIO = 10**4
```

[G-02] Do not initialize variables with default values

```
211: bool constant FEE_DOWN = false;
```

[G-03] The result of function calls should be cached rather than re-calling the function

Function `t(self)` is being called 3 times. Firstly, in the `if` -condition (line 98), then - when condition fails: `ABDK_ONE.sub(t(self))` and `self.px_final.mul(t(self))`.

```
File: EvolvingProteus.sol

097:     function p_min(Config storage self) public view returns
098:         if (t(self) > ABDK_ONE) return self.px_final;
099:         else return self.px_init.mul(ABDK_ONE.sub(t(self)))
100:     }
```

The same issue occurs for `p_max`:

```
File: EvolvingProteus.sol

106:     function p_max(Config storage self) public view returns
107:         if (t(self) > ABDK_ONE) return self.py_final;
108:         else return self.py_init.mul(ABDK_ONE.sub(t(self)))
109:     }
```

The result of `t(self)` should be cached in a variable, instead of being executed 3 times.

[G-04] More likely to revert operations should be executed first

```
File: EvolvingProteus.sol

250:         // price value checks
251:         if (py_init >= MAX_PRICE_VALUE || py_final >= MAX_P
252:         if (px_init <= MIN_PRICE_VALUE || px_final <= MIN_P
253:
254:         // at all times x price should be less than y price
255:         if (py_init <= px_init) revert InvalidPrice();
256:         if (py_final <= px_final) revert InvalidPrice();
```

Since `py_init`, `px_init`, `py_final` and `px_final` are `int128`, there are `2**128` values they can be assigned to.

This implies, that condition `if (py_init <= px_init)` will revert for `(2**128) * (2**128 - 1) / 2 + 2**128` cases, while condition `if (py_init >= MAX_PRICE_VALUE || py_final >= MAX_PRICE_VALUE)` will revert for `(2**128 - MAX_PRICE_VALUE) * 2**128 + (2**128 - MAX_PRICE_VALUE) * 2**128`.

According to that, reverts for `py_init <= px_init` and `py_final <= px_final` are more likely than reverts for `py_init >= MAX_PRICE_VALUE || py_final >= MAX_PRICE_VALUE` and `px_init <= MIN_PRICE_VALUE || px_final <= MIN_PRICE_VALUE`. Thus, ordering of lines 251-252 and 255-256 should be changed:

```
if (py_init <= px_init) revert InvalidPrice();
if (py_final <= px_final) revert InvalidPrice();

if (py_init >= MAX_PRICE_VALUE || py_final >= MAX_PRICE_VALUE)
if (px_init <= MIN_PRICE_VALUE || px_final <= MIN_PRICE_VALUE)
```

[G-05] OR in `if` -condition can be rewritten to two single `if` conditions

```
File: EvolvingProteus.sol
251:         if (py_init >= MAX_PRICE_VALUE || py_final >= MAX_P
252:         if (px_init <= MIN_PRICE_VALUE || px_final <= MIN_P
```

Refactoring the `if` -condition in a way it won't be containing the `||` operator will save more gas.

This is a simple forge test, which demonstrates gas usage for both versions:

```
    function testIfWithOR(
        int128 py_init,
        int128 px_init,
        int128 py_final,
        int128 px_final,
        uint256 duration
    ) public returns (uint) {

        if (py_init >= MAX_PRICE_VALUE || py_final >= MAX_PRICE_
        if (px_init <= MIN_PRICE_VALUE || px_final <= MIN_PRICE_
    }
   function testIfWithoutOr(
        int128 py_init,
        int128 px_init,
        int128 py_final,
        int128 px_final,
        uint256 duration
    ) public returns (uint) {

        if (py_init >= MAX_PRICE_VALUE) return 1;
        if (py_final >= MAX_PRICE_VALUE) return 1;

        if (px_init <= MIN_PRICE_VALUE) return 1;
        if (px_final <= MIN_PRICE_VALUE) return 1;
    }
```

```
Gas:testIfWithOR(int128,int128,int128,int128,uint256):(uint256)
```

```
Gas:testIfWithoutOr(int128,int128,int128,int128,uint256):(uint25
```

This suggests, that conditions should be rewritten to:

```
if (py_init >= MAX_PRICE_VALUE) revert MaximumAllowedPriceExceed
if (py_final >= MAX_PRICE_VALUE) revert MaximumAllowedPriceExce
if (px_init <= MIN_PRICE_VALUE) revert MaximumAllowedPriceExceed
if (px_final <= MIN_PRICE_VALUE) revert MaximumAllowedPriceExce
```

[G-06] Incorrect usage of `ABDKMath64x64.divu` in constructor

```
259:         if (py_init.div(py_init.sub(px_init)) > ABDKMath64x6
260:         if (py_final.div(py_final.sub(px_final)) > ABDKMath6
```

Function `MAX_PRICE_RATIO` is declared as `int256 constant MAX_PRICE_RATIO = 10**4;` and it is only used in the `constructor`. Having this constant declared as `int256` and then being cast to `uint` is an unreasonable waste of gas.

Declare this constant as uint and do not cast it later:

```
201:         uint constant MAX_PRICE_RATIO = 10**4

(...)

259:         if (py_init.div(py_init.sub(px_init)) > ABDKMath64x6
260:         if (py_final.div(py_final.sub(px_final)) > ABDKMath6
```

[G-07] Unnecessary double `if` condition in `_swap` and `_lpTokenSpecified`

```
File: EvolvingProteus.sol
```

```
529:             if (specifiedToken == SpecifiedToken.X) {
530:                 int256 fixedPoint = xi + roundedSpecifiedAm
531:                 (xf, yf) = _findFinalPoint(
532:                     fixedPoint,
533:                     utility,
534:                     _getPointGivenXandUtility
535:                 );
536:             } else {
537:                 int256 fixedPoint = yi + roundedSpecifiedAm
538:                 (xf, yf) = _findFinalPoint(
539:                     fixedPoint,
540:                     utility,
541:                     _getPointGivenYandUtility
542:                 );
543:             }
544:         }
545:
546:         // balance checks with consideration the computed a
547:         if (specifiedToken == SpecifiedToken.X) {
548:             computedAmount = _applyFeeByRounding(yf - yi, f
549:             _checkBalances(xi + specifiedAmount, yi + compu
550:         } else {
551:             computedAmount = _applyFeeByRounding(xf - xi, f
552:             _checkBalances(xi + computedAmount, yi + specif
553:         }
```

Function `_swap` checks `specifiedToken == SpecifiedToken.X` condition twice: firstly in line 529, then in line 547. There are no additional operations within those two `if` blocks, thus, the result of `specifiedToken == SpecifiedToken.X` will be always the same. This implies that the 2nd `if` condition can be removed and lines 548-549 can be included in the first `if` block:

```
            int256 utility = _getUtility(xi, yi);

            if (specifiedToken == SpecifiedToken.X) {
                int256 fixedPoint = xi + roundedSpecifiedAmount;
                (xf, yf) = _findFinalPoint(
                    fixedPoint,
                    utility,
                    _getPointGivenXandUtility
                );

                computedAmount = _applyFeeByRounding(yf - yi, f
```

```
            _checkBalances(xi + specifiedAmount, yi + comput
        } else {
            int256 fixedPoint = yi + roundedSpecifiedAmount;
            (xf, yf) = _findFinalPoint(
                fixedPoint,
                utility,
                _getPointGivenYandUtility
            );
             computedAmount = _applyFeeByRounding(xf - xi, f
            _checkBalances(xi + computedAmount, yi + specifi
        }
```

The same issue occurs for `_lpTokenSpecified`:

```
File: EvolvingProteus.sol

626:          if (specifiedToken == SpecifiedToken.X)
627:              (xf, yf) = _findFinalPoint(
628:                  yi,
629:                  uf,
630:                  _getPointGivenYandUtility
631:              );
632:          else
633:              (xf, yf) = _findFinalPoint(
634:                  xi,
635:                  uf,
636:                  _getPointGivenXandUtility
637:              );
638:
639:          // balance checks with consideration the computed a
640:          if (specifiedToken == SpecifiedToken.X) {
641:              computedAmount = _applyFeeByRounding(xf - xi, f
642:              _checkBalances(xi + computedAmount, yf);
643:          } else {
644:              computedAmount = _applyFeeByRounding(yf - yi, f
645:              _checkBalances(xf, yi + computedAmount);
646:          }
```

[G-08] Unnecessary variable declaration in `_reserveTokenSpecified`

File: EvolvingProteus.sol

```
589:          uint256 result = Math.mulDiv(uint256(uf), uint256(s
590:          require(result < INT_MAX);
591:          int256 sf = int256(result);
592:
593:          // apply fee to the computed amount
594:          computedAmount = _applyFeeByRounding(sf - si, feeDi
```

Declaring the `int256 sf` variable is redundant. Since it's only being used once (line 594), `int256(result)` can be used directly: `computedAmount = _applyFeeByRounding(int256(result) - si, feeDirection);`.

[G-09] Calculations which won't underflow can be `unchecked`

```
834:          if (absoluteValue < FIXED_FEE * 2) revert AmountErro
835:
836:          uint256 roundedAbsoluteAmount;
837:          if (feeUp) {
838:              roundedAbsoluteAmount =
839:                  absoluteValue +
840:                  (absoluteValue / BASE_FEE) +
841:                  FIXED_FEE;
842:              require(roundedAbsoluteAmount < INT_MAX);
843:          } else
844:              roundedAbsoluteAmount =
845:                  absoluteValue -
846:                  (absoluteValue / BASE_FEE) -
847:                  FIXED_FEE;
```

According to line 834, `absoluteValue` is at least `>= 2 * FIXED_FEE` (otherwise, the function will revert with `AmountError()`). This implies, that lines 844-847 will never underflow.

In the worst case scenario, `absoluteValue = 2 * FIXED_FEE` (because `if (absoluteValue < FIXED_FEE * 2)`, then the function will revert). Moreover, we know that `BASE_FEE = 800` (line 186).

```
roundedAbsoluteAmount = absoluteValue - (absoluteValue / BASE_FE

for BASE_FEE = 800

roundedAbsoluteAmount = absoluteValue - (absoluteValue / 800) -

for absoluteValue = 2 * FIXED_FEE

2 * FIXED_FEE - (2 * FIXED_FEE  / 800) - FIXED_FEE = FIXED_FEE -

FIXED_FEE - (2 * FIXED_FEE  / 800) >= 0
800 * FIXED_FEE - 2 * FIXED_FEE >= 0
798 * FIXED_FEE >= 0
```

This implies that calculations won't underflow and can be inserted into the `unchecked` block:

```
unchecked{roundedAbsoluteAmount =
              absoluteValue -
              (absoluteValue / BASE_FEE) -
              FIXED_FEE;}
```

[G-10] Pre-calculate equations which contain only constant values in `constructor`

```
259:         if (py_init.div(py_init.sub(px_init)) > ABDKMath64x6
260:         if (py_final.div(py_final.sub(px_final)) > ABDKMath6
```

Since `MAX_PRICE_RATIO` is a constant value, the value of `ABDKMath64x64.divu(uint(MAX_PRICE_RATIO), 1))` can be calculated before compiling the contract. Calling `ABDKMath64x64.divu` on a known, constant value is a waste of gas.

[G-11] Pre-calculate equations which contains only constant values in `_getUtility`

```
709:         int128 two = ABDKMath64x64.divu(uint256(2 * MULTIPL
710:         int128 one = ABDKMath64x64.divu(uint256(MULTIPLIER),
```

Since `MULTIPLIER` is a constant value, the values of `ABDKMath64x64.divu(uint256(2 * MULTIPLIER), uint256(MULTIPLIER))` and `ABDKMath64x64.divu(uint256(MULTIPLIER), uint256(MULTIPLIER))` can be calculated before compiling the contract. Calling `ABDKMath64x64.divu` on a known, constant value is a waste of gas.

[G-12] Some equations can be inlined to reduce number of variables declarations

`_getUtility`

```
    int128 two = ABDKMath64x64.divu(uint256(2 * MULTIPLIER), uint256
    int128 one = ABDKMath64x64.divu(uint256(MULTIPLIER), uint256(MUI

    int128 aQuad = (a.mul(b).sub(one));
    int256 bQuad = (a.muli(y) + b.muli(x));
    int256 cQuad = x * y;

    int256 disc = int256(Math.sqrt(uint256((bQuad**2 - (aQuad.muli(c
    int256 r0 = (-bQuad*MULTIPLIER + disc*MULTIPLIER) / aQuad.mul(tw
    int256 r1 = (-bQuad*MULTIPLIER - disc*MULTIPLIER) / aQuad.mul(tw
```

Can be changed to:

```
    int128 two = ABDKMath64x64.divu(uint256(2 * MULTIPLIER), uint256

    int128 aQuad = (a.mul(b).sub(ABDKMath64x64.divu(uint256(MULTIPL
    int256 bQuad = (a.muli(y) + b.muli(x));

    int256 disc = int256(Math.sqrt(uint256((bQuad**2 - (aQuad.muli(x
    int256 r0 = (-bQuad*MULTIPLIER + disc*MULTIPLIER) / aQuad.mul(tw
    int256 r1 = (-bQuad*MULTIPLIER - disc*MULTIPLIER) / aQuad.mul(tw
```

`_getPointGivenXandUtility`

```
int256 a_convert = a.muli(MULTIPLIER);
int256 b_convert = b.muli(MULTIPLIER);
x0 = x;

int256 f_0 = ((( x0  * MULTIPLIER ) / utility) + a_convert);
int256 f_1 = ((MULTIPLIER * MULTIPLIER / f_0) -  b_convert);
int256 f_2 = (f_1 * utility) / MULTIPLIER;
y0 = f_2;
```

Can be changed to:

```
x0 = x;
y0 = (((MULTIPLIER * MULTIPLIER / ((( x0  * MULTIPLIER ) / utili
```

`_getPointGivenYandUtility`

```
int256 a_convert = a.muli(MULTIPLIER);
int256 b_convert = b.muli(MULTIPLIER);
y0 = y;

int256 f_0 = (( y0  * MULTIPLIER ) / utility) + b_convert;
int256 f_1 = ( ((MULTIPLIER)*(MULTIPLIER) / f_0) - a_convert );
int256 f_2 = (f_1 * utility) / (MULTIPLIER);
x0 = f_2;
```

Can be changed to:

```
y0 = y;
x0 = (( ((MULTIPLIER)*(MULTIPLIER) / (( y0  * MULTIPLIER ) / uti
```

[G-13] `_getUtility` calculates the same value twice

```
717:        int256 r0 = (-bQuad*MULTIPLIER + disc*MULTIPLIER) /
718:        int256 r1 = (-bQuad*MULTIPLIER - disc*MULTIPLIER) /
```

Cache results of `-bQuad*MULTIPLIER`, `disc*MULTIPLIER` and `aQuad.mul(two).muli(MULTIPLIER)` to not calculate it twice.

viraj124 (Shell) acknowledged

Audit Analysis

For this audit, 11 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The report highlighted below by **0xSmartContract** received the top score from the judge.

The following wardens also submitted reports: Udsen, catellatech, ihtishamsudo, oakcobalt, carrotsmuggler, rjs, 0xmystery, plainshift, pfapostol, *and* moneyversed.

Summary

Head	Details
The approach I followed when reviewing the code	Stages in my code review and analysis
Analysis of the code base	What is unique? How are the existing patterns used?
Test analysis	Test scope of the project and quality of tests
Centralization risks	How was the risk of centralization handled in the p project, what could be alternatives?
Systemic risks	Potential systemic risks in the project
Competition analysis	What are similar projects?
Security Approach of the Project	Audit approach of the Project
Other Audit Reports and Automated Findings	What are the previous Audit reports and their analysis
Gas Optimization	Gas usage approach of the project and alternative solutions to it
Other recommendations	What is unique? How are the existing patterns used?
New insights and learning from this audit	Things learned from the project

The approach I followed when reviewing the code

First, by examining the scope of the code, I determined my code review and analysis strategy for the **Shell Protocol**.

I analyzed and audited the subject in the following steps:

Stage	Details	Information
Compile and Run Test	Installation	Test and installation structure is simple, cleanly designed.
Architecture Review	The Proteus-AMM explainer provides a high-level overview of the Project system and then describe the core components.	Provides a basic architectural teaching for General Architecture.
Graphical Analysis	Graphical Analysis with Solidity-metrics	A visual view has been made to dominate the general structure of the codes of the project.
Slither Analysis	Slither Report	The project does not currently have a slither result, a slither control was created from scratch.
Test Suits	Tests	In this section, the scope and content of the tests of the project are analyzed.
Manual Code Review	Scope	Top-down analysis of codes according to architectural design, IDE used: VsCode.
Project White Paper	WhitePaper	
Infographic	Figma	I made Visual drawings to understand the hard-to-understand mechanisms.
Special focus on Areas of Concern	Areas of Concern	Evolving Proteus's algorithm has six parameters that determine liquidity concentration.

Analysis of the code base

Image #1:

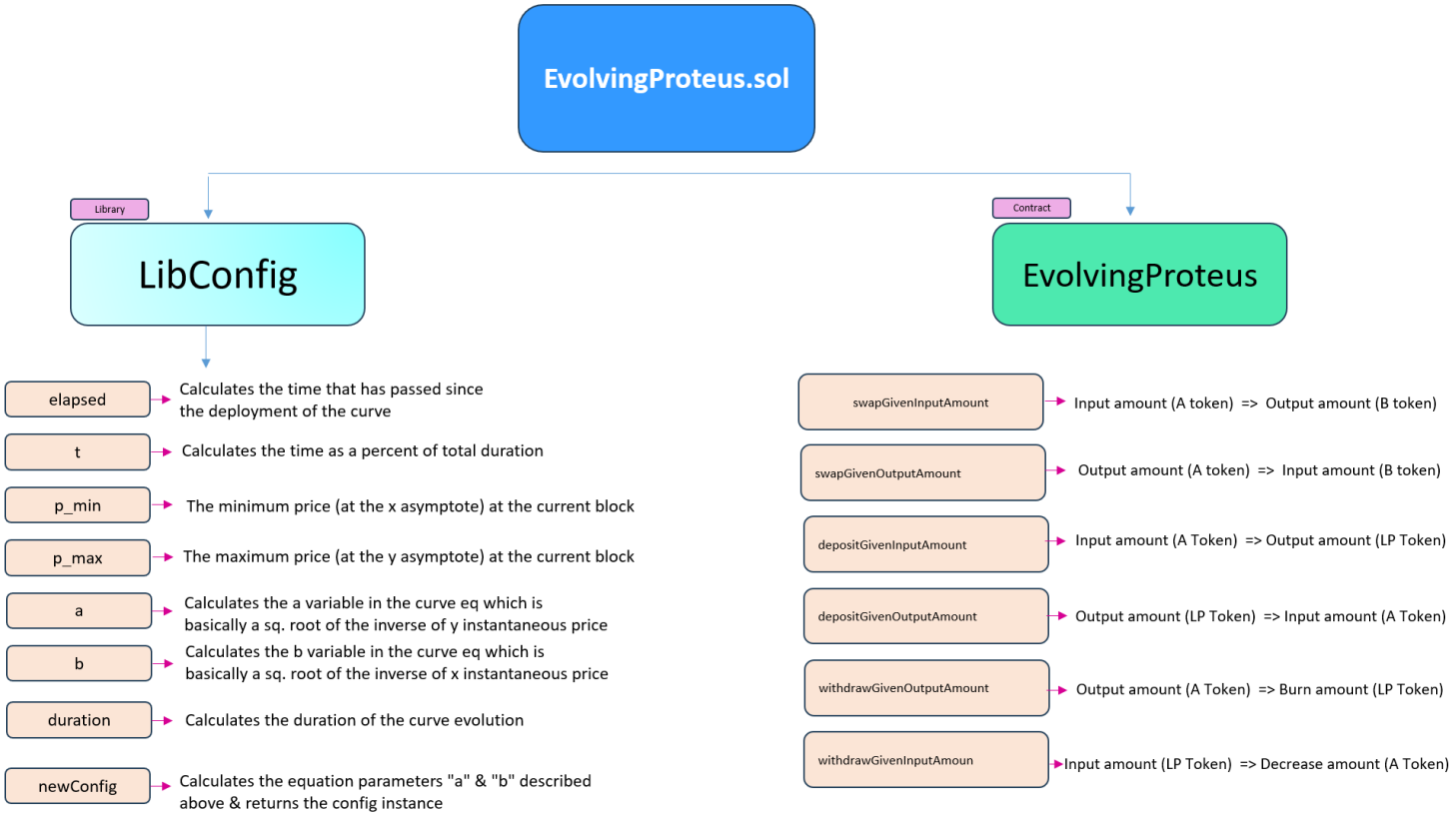
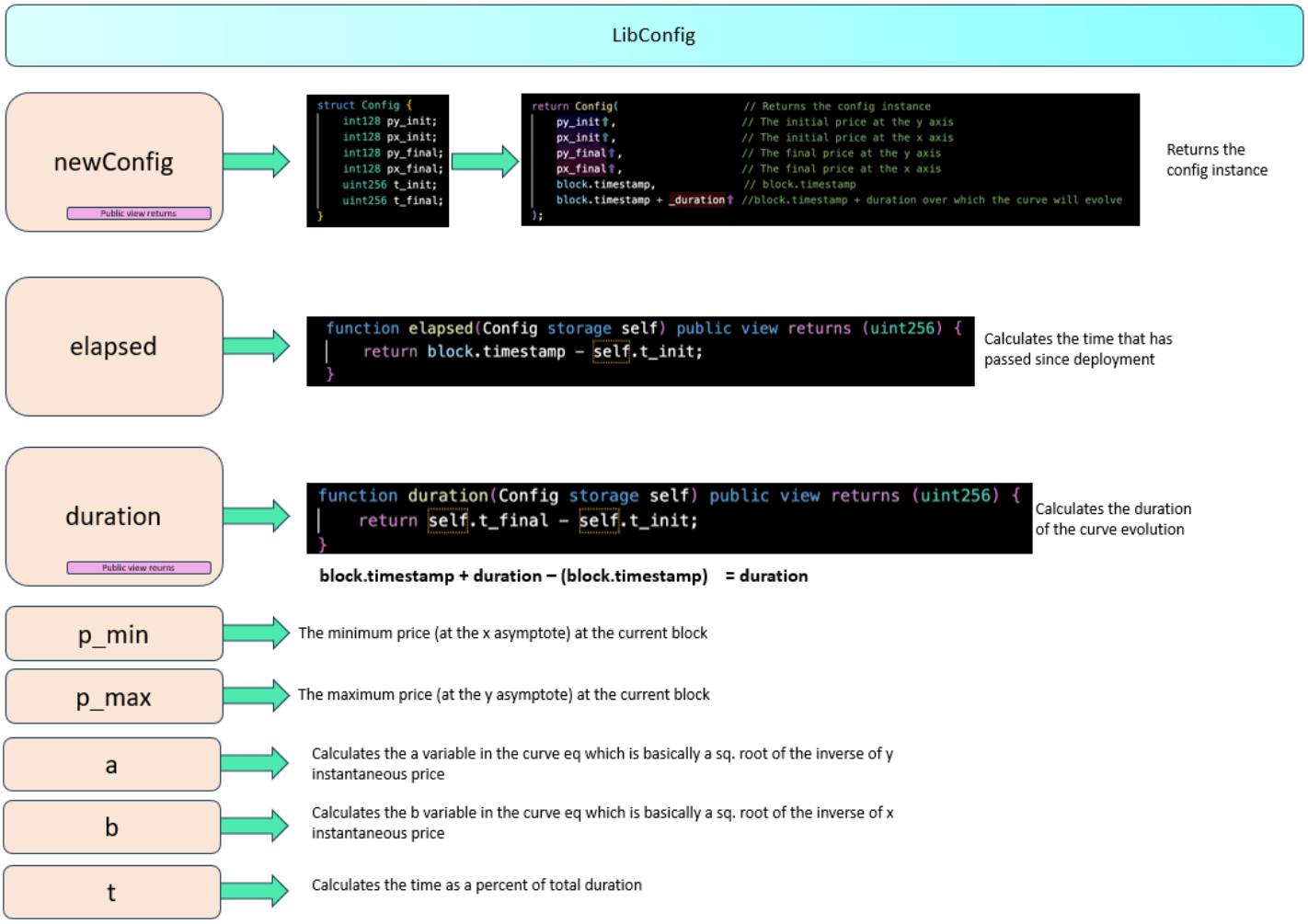


Image #2:



Test analysis

Note: image has been removed due to an outdated link

ForkEvolvingProteus.t.sol :

- Invariant tests can be great tools for shaking out invalid assumptions, complex edge cases, and unexpected interactions in a smart contract system. But it can also be challenging to channel the fuzzer's unconstrained chaos into a suite of meaningful, reliable tests.

Evolving Proteus Invariants test list (ForkEvolvingProteus.t.sol):

Number	Head	Test Details
1.	Token balance check	The differences in pool token balances after a swap is same as the differences in user token balances after factoring in the fee.
2.	Token supply check	Utility & utility/lp token supply does not decrease after swap.
3.	Deposit check	For a deposit, utility will always increase, and util/lp does not decrease.
4.	Withdraw check	For a withdrawl, utility will always decrease, and util/lp does not decrease.
5.	Soft invariant	x price decreases over time & y price stays constant.
6.	Soft invariant	when x price increases over time & y price stays constant.
7.	Soft invariant	when y price decreases over time & x price stays constant.
8.	Soft invariant	when y price increases over time & x price stays constant.
9.	Soft invariant	when x & y prices both increase with time.
10.	Soft invariant	when x & y prices both decrease with time.
11.	Soft invariant	when x & y prices both stay constant.

Invariant tests recommended to be added (ForkEvolvingProteus.t.sol):

Number	Head	Test Details	
1.	Soft invariant	when x price increases over time & y price decreases.	
2.	Soft invariant	when y price increases over time & x price decreases.	

Centralization risks

There is no centrality risk in the controlled scope.

Systemic risks

We can talk about a systemic risk here because there are some Layer2 special cases.

If Arbitrum is compile **Compatible** with 0.8.20 and newer, contracts compiled with these versions will result in a non-functional or potentially damaged version that does not behave as expected. The default behavior of the compiler will be to use the latest version which means it will compile with version 0.8.20 which will produce broken code by default.

According to the given code, this problem does not exist, but it should be taken care of while deploying.

Competition analysis

Other projects with a similar structure to the project:

1. dfx finance
2. Gamma
3. Osmosis

However, with its mathematical modeling, the Shell Protocol is unique.

Security Approach of the Project

Successful current security understandings of the project:

1. The most important security point of Shell protocol is contracts are non-custodial, meaning NOBODY (not even the core team or the Shell DAO) is able to access the funds held in any of the core contracts.

2. Proteus Pool Safeguards System; a pool's deployer can assign a wallet the ability to freeze the pool in case of an emergency. LPs may still withdraw their tokens, but swaps and deposits are disabled. The Shell core team controls a 2/3 multi-signature wallet, the `ShellMultiSig`, that can freeze trading on Proteus pools assigned to it.
3. On-Chain Monitoring System; Shell conducts on-chain security monitoring with Forta. Anyone can subscribe to updates from the Shell Forta bot. This bot tracks Shell transactions in which wrapping, unwrapping, swapping, depositing, or withdrawals occur over a threshold amount. If transactions occur with unusually high token amounts, the bot sends out an alert.

   https://app.forta.network/bot/0x7f9afc392329ed5a473bcf304565adf9c2588ba4bc060f7d215519005b8303e3

4. First, they created the main audit from a reputable auditing organization like Trail of Bits and resolved all the security concerns in the report.

   https://github.com/trailofbits/publications/blob/master/reviews/ShellProtocolv2.pdf

5. They manage the 2nd audit process with an innovative audit such as Code4rena, in which many auditors examine the codes.
6. They set the `$` 100,000 ImmuneFi reward.

What the project should add in the understanding of Security:

1. By distributing the project to testnets, this ensures that the audits are carried out in an on-chain audit (this will increase coverage).
2. After the project is published on the mainnet, there should be emergency action plans (not found in the documents).
3. No pause mechanism; this is a chaotic situation, which can be thought of as a choice between decentralization and security. Having a pause mechanism makes sense in order not to damage user funds in case of a possible problem in the project.
4. No upgradability; there are use cases of the upgradable pattern in defi projects using mathematical models, but it is a design and security option.

Other Audit Reports and Automated Findings

Especially low detections in the Automated Finding Report should be taken into account.

Other Audit Reports (TrailOfBits):

Here is a summary of the issues found in the audit report with Exposure Analysis and Category details:

Audit Report TrailOfBits

EXPOSURE ANALYSIS

Severity	Count
High	4
Medium	2
Low	0
Informational	2
Undetermined	0

CATEGORY BREAKDOWN

Category	Count
Access Controls	1
Auditing and Logging	1
Data Validation	2
Patching	1
Testing	1
Timing	1
Undefined Behavior	1

Gas Optimization

The project is generally efficient in terms of gas optimizations, many generally accepted gas optimizations have been implemented. Gas optimizations with minor effects are already mentioned in automatic finding, but gas optimizations will not be a priority considering code readability and codebase size.

Gas Optimization Recommendations

1. The highest gas expenditure in the project occurs when creating pool instances. If the architecture here was designed like the singleton architecture in UniswapV4, gas savings will be close to 90%.

2. The most important gas usage of the contract subject to the audit is the `ABDKMath64x64.sol` library that it uses. More gas optimized by [prb-math] may be preferred. Detailed gas comparisons are available on the project's github link

PRBMath

Gas estimations based on the [v2.0.1] and the [v3.0.0] releases.

SD59x18	Min	Max	Avg		UD60x18	Min	Max	Avg
abs	68	72	70		n/a	n/a	n/a	n/a
avg	95	105	100		avg	57	57	57
ceil	82	117	101		ceil	78	78	78
div	431	483	451		div	205	205	205
exp	38	2797	2263		exp	1874	2742	2244
exp2	63	2678	2104		exp2	1784	2652	2156
floor	82	117	101		floor	43	43	43
frac	23	23	23		frac	23	23	23
gm	26	892	690		gm	26	893	691
inv	40	40	40		inv	40	40	40
ln	463	7306	4724		ln	419	6902	3814
log10	104	9074	4337		log10	503	8695	4571
log2	377	7241	4243		log2	330	6825	3426
mul	455	463	459		mul	219	275	247
pow	64	11338	8518		pow	64	10637	6635
powu	293	24745	5681		powu	83	24535	5471
sqrt	140	839	716		sqrt	114	846	710

ABDKMath64x64

Gas estimations based on the v3.0 release of ABDKMath. See my [abdk-gas-estimations] repo.

Method	Min	Max	Avg
abs	88	92	90
avg	41	41	41
div	168	168	168
exp	77	3780	2687
exp2	77	3600	2746
gavg	166	875	719
inv	157	157	157
ln	7074	7164	7126
log2	6972	7062	7024
mul	111	111	111
pow	303	4740	1792
sqrt	129	809	699

Other recommendations

- Logic similar to the singleton pattern similar to Uniswapv4 can be used in a project. This is a gas-optimized and innovative solution (expressed for the overall project, not for the audit contract).
- The use of assembly in project codes is very low. I especially recommend using such useful and gas-optimized code patterns:

  https://github.com/dragonfly-xyz/useful-solidity-patterns/tree/main/patterns/assembly-tricks-1

- It is seen that the latest versions of imported important libraries such as Openzeppelin are not used in the project codes; it should be noted.

  https://security.snyk.io/package/npm/@openzeppelin%2Fcontracts/

- A good model can be used to systematically assess the risk of the project, for example, this modeling is recommended:

  https://www.notion.so/Smart-Contract-Risk-Assessment-3b067bc099ce4c31a35ef28b011b92ff#7770b3b385444779bf11e677f16e101

e

Here are my other suggestions for corrections, the details of which are in my QA Report.

- L-01 - `result` require block is inconsistent with NatSpec comment
- L-02 - `libusb` dependency can be a security risk
- L-03 - There is a difference in the formula with the NatSpec comments
- L-04 - There may be problems with the use of Arbitrum
- N-01 - Project Upgrade and Stop Scenario should be
- N-02 - Missing Event for initialize

New insights and learning from this audit

- `ForkEvolvingProteus.t.sol` invariant tests are extensive, quality tests in a high-mathematical computational project and are remarkable. The project developers did a great job!
- I learned, in depth, the usage and details of `ABDKMath64x64.sol` library.

Time spent:

13 hours

0xRobocop (lookout) commented:

> In my point of view, although the report does not touch a core area of concern specific to the code like Issue #69, the report is pretty complete in terms of security, as a holistic approach.

viraj124 (Shell) acknowledged and commented:

> A few observations:

- Regarding the invariant not testing when prices move in opposite directions, the reason for that is the behaviour is not consistent. Hence, we kept that out of scope for the audit and we won't be using that price configuration in the near future.

- Regarding singleton behaviour, all our accounting for all different pools is done in the ocean, so that compensates for deploying different contracts for each pool. That being said, we might change that in the future if needed. For now, due to the architecture of the ocean and for us to support more defi primitives like lending and option markets and not just be limited to amm's, we decided not to go with the singleton pool deployment approach.

Disclosures

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

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

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

Top