



# SMART CONTRACT AUDIT

**ZOKYO.**

July 7th, 2021 | v. 1.0

## **PASS**

Zokyo's Security Team has concluded that this smart contract passes security qualifications to be listed on digital asset exchanges



# TECHNICAL SUMMARY

This document outlines the overall security of the Polars smart contracts, evaluated by Zokyo's Blockchain Security team.

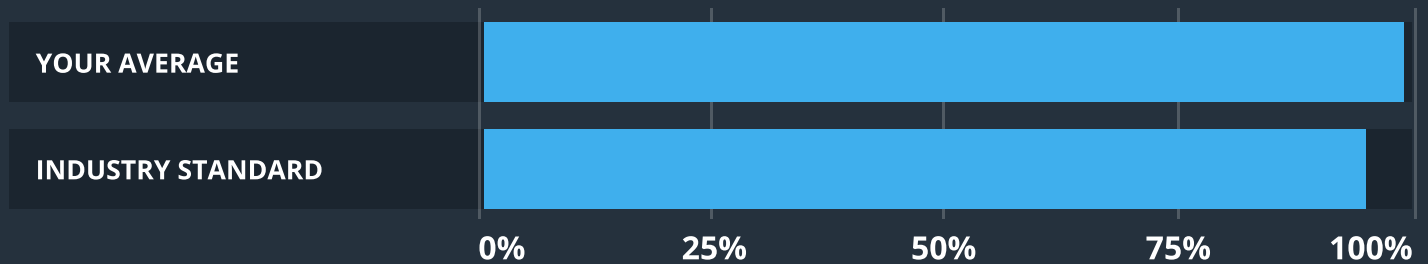
The scope of this audit was to analyze and document the Polars smart contract codebase for quality, security, and correctness.

## Contract Status



There were 3 critical issues found during the audit which were successfully resolved.

## Testable Code



The testable code is 99.07%, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract, rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that's able to withstand the Ethereum network's fast-paced and rapidly changing environment, we at Zokyo recommend that the Polars team put in place a bug bounty program to encourage further and active analysis of the smart contract.

# TABLE OF CONTENTS

- Auditing Strategy and Techniques Applied . . . . . 3
- Summary . . . . . 4
- Structure and Organization of Document . . . . . 5
- Manual Review . . . . . 6
- Code Coverage and Test Results for all files . . . . . 22
  - Tests written by Polars team . . . . . 22
  - Tests written by Zokyo Secured team . . . . . 26

# AUDITING STRATEGY AND TECHNIQUES APPLIED

The Smart contract's source code was taken from the Polars repository – [https://github.com/yurivin/Polars\\_Zokyo/commit/d1bd01c000fdc37627eb5dd72ce70eb4b289481c](https://github.com/yurivin/Polars_Zokyo/commit/d1bd01c000fdc37627eb5dd72ce70eb4b289481c)

Last commit – cb1a81056390857a76f2a15500f1ac3fc57afe7a

## Throughout the review process, care was taken to ensure that the token contract:

- Implements and adheres to existing Token standards appropriately and effectively;
- Documentation and code comments match logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices in efficient use of gas, without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the latest vulnerabilities;
- Whether the code meets best practices in code readability, etc.

Zokyo's Security Team has followed best practices and industry-standard techniques to verify the implementation of Polars smart contracts. To do so, the code is reviewed line-by-line by our smart contract developers, documenting any issues as they are discovered. Part of this work includes writing a unit test suite using the Truffle testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

1	Due diligence in assessing the overall code quality of the codebase.	3	Testing contract logic against common and uncommon attack vectors.
2	Cross-comparison with other, similar smart contracts by industry leaders.	4	Thorough, manual review of the codebase, line-by-line.

## SUMMARY

Within the scope of this audit, Zokyo auditing team has reviewed and investigated the contracts given in order to find vulnerabilities, inefficient gas usage, and operational issues. All the findings are described in this audit report.

There were 3 critical issues found during the audit. Polars dev team has resolved all of them. Besides critical issues found, Zokyo auditors have found a couple of more issues with high, medium, and low levels of severity. Most of these issues were successfully fixed and bear no risk for the end-user.

Based on the findings and the second review of the fixes made by the Polars team, we can give a score of 95 to the contracts given.

It is worth mentioning that the contracts are well written and structured and most of the findings during the audit have no impact on contract performance.

## STRUCTURE AND ORGANIZATION OF DOCUMENT

For ease of navigation, sections are arranged from most critical to least critical. Issues are tagged “Resolved” or “Unresolved” depending on whether they have been fixed or addressed. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

 **Critical**

The issue affects the ability of the contract to compile or operate in a significant way.

 **High**

The issue affects the ability of the contract to compile or operate in a significant way.

 **Medium**

The issue affects the ability of the contract to operate in a way that doesn't significantly hinder its behavior.

 **Low**

The issue has minimal impact on the contract's ability to operate.

 **Informational**

The issue has no impact on the contract's ability to operate.

# MANUAL REVIEW

## User can't get governance token unless he is the one who initially provided it

CRITICAL | RESOLVED

Let's assume User A put a governance token and got his incentive tokens. He sends some of them to User B. But User B can't exchange them back to governance, since contract incentives.sol expects user A. For example in function putIncentivesTokens.

### Recommendation:

If this is expected behaviour (incentive tokens aren't supposed to be shared), then please add a comment that briefly explains why and how the token is supposed to be used.

But if there is no reason for sending incentives tokens anywhere but back to the incentive contract by the same person who got them.. Then maybe you don't even have to create a new token. Simple record in incentives contract is enough.

### Partner response for this issue:

Incentives tokens can be shared for their own purposes because they provide the possibility to receive rewards in other networks. So this is why we need it. The point that users cannot withdraw more governance tokens from the incentives contract than they put there is just a part of the logic we expect.

Users can trade Incentives tokens if they want to receive more rewards from specific platform instance, that is why there is a sense to exchange and share it.

Users may change balance of their incentives tokens to get more rewards but later, when they want to withdraw governance tokens they can exchange incentives tokens back.

### Re-audit:

Fixed.



## User can get Incentives tokens for free

CRITICAL | RESOLVED

At contract incentives.sol function `withdrawIncentives` accepts `tokenAddress` and doesn't check if the token is actually an incentive one. So, users can pass, for example, an address of the governance token and get it back without `withdrawGovernance` function.

After withdrawal governance, the user can call `withdrawAllIncentives()` (because his `govTokenBalances` is still positive) and get every incentive token for free.

Additionally, If a user notices any token, which price > than the governance one (WBTC for example, if someone sent it on incentives contract by mistake), he can "exchange" if for governance token using same `withdrawIncentives` function.

### Recommendation:

Check if `tokenAddress` is expected incentive token address.

### Re-audit:

Fixed.

## Incentives contract does not compile

CRITICAL | RESOLVED

Contract does not compile because it doesn't implement the function `withdrawAllIncentives`. Also some methods mark `override` but they don't override anything.

### Recommendation:

Implement function `withdrawAllIncentives()` `external override {}`, or delete it from interface `IIncentives`.

Delete `override` mark on methods: `getGovBalances`, `getMaxBorrowed`, `lockGovernance`, `setCrowdsale`.

### Re-audit:

Fixed.

## Function “buyBack” can be called by anyone

HIGH | RESOLVED

At contract PrimaryPoolERC20.sol function buyBack allows to buy back tokens for any user. This will work if there is permission on erc20 token transfer. Although tokens can't be stolen this way, but it may lead to unwanted “buy backs”.

### Recommendation:

Unless there is a special requirement, please replace

```
function buyBack(address destination, uint256 tokensAmount) public {
    require(destination != address(0), "Destination address is empty");
    ...
    _collateralization.buyBack(destination, tokensAmount, collateralAmount);
    emit BuyBack(tokensAmount, blackAndWhitePrice);
}
```

with, for example this:

```
function buyBack(uint256 tokensAmount) public {
    ...
    _collateralization.buyBack(msg.sender, tokensAmount, collateralAmount);
    emit BuyBack(tokensAmount, blackAndWhitePrice);
}
```

In case buyBack indeed requires transferring of tokens from another address for user, please add verification of the approved addresses.

### Re-audit:

Fixed.

## Governance can replace pool and buyBack tokens for unexpected price

MEDIUM | UNRESOLVED

At contracts `CollateralizationERC20.sol`, `PrimaryCollateralizationERC20.sol` there are 4 functions that can be called only by pool: `buy`, `buySeparately`, `buyBack`, `buyBackSeparately`.

This is good, since the functions accept both amounts: `tokensAmount` and `payment` (and don't check if price is fair).

At the same time, there is possibility of updating the pool (by governance, function `changePoolAddress`), and withdraw lots of collateral token for very small payment.

### **Recommendation:**

Remove `changePoolAddress` function (if possible). If not, add a comment that briefly explains how all funds are secured. For example: governance address will be a contract that requires voting for pool update.

### **Re-audit:**

Skipped.

## Functions may become unexecutable due to potentially huge amount of allTokens array

MEDIUM | RESOLVED

At contract Incentives.sol there are 4 functions with this cycle: withdrawAllIncentives, putIncentivesTokens, userState, availableTokens.

If the array of tokens becomes significant in size (more than 500, need to perform additional tests to get exact array size), first it will take a lot of gas to execute, then it will hit gas limits.

### Recommendation:

If no significant amount of tokens is expected (less than 100 for example), add a comment that confirms this. If more than 100, additional tests are required to develop a suitable solution.

### Partner response for this issue:

In possible solutions Zokyo auditors write that there is no issue if there will be less than 100 tokens in the array. I commented out that we will not have 50 tokens and most probably there will be at max 10-20 tokens according to the number of blockchain networks that we will use. This number is much less than 100 tokens border provided by your auditors. But I see that issue is still unresolved even if I have provided an expected response.

### Re-audit:

Fixed.

## Unnecessary variables declaration

LOW | UNRESOLVED

At contract Bpool.sol at function getNormalizedWeight there is a denorm variable that is actually used only once and can be removed.

### Recommendation:

Use direct insertion, example:

```
return bdiv(_records[token].denorm, _totalWeight);
```

Worth to mention, that not all variables have to be replaced in such a manner. Some of them indeed improve transparency of code (example: oldWeight variable in rebind function). But overall, where possible, use direct variable/function insertion (example: joinPool function):

```
uint poolTotal = totalSupply();  
uint ratio = bdiv(poolAmountOut, poolTotal);
```

Still perfectly readable if we use:

```
uint ratio = bdiv(poolAmountOut, totalSupply()); //totalSupply = poolTotal
```

### Re-audit:

Skipped.

## Unnecessary variables declaration

LOW | RESOLVED

Although the current code looks good, it can be a little bit more gas efficient. At contract `CollateralizationERC20.sol` at function `getStoredTokensAmount` there are 2 new variables: `whiteTokensAmount` and `blackTokensAmount`. But it isn't necessarily to do so just for returning the variable.

### Recommendation:

Replace

```
uint256 whiteTokensAmount = _whiteToken.balanceOf(address(this));
uint256 blackTokensAmount = _blackToken.balanceOf(address(this));
return (whiteTokensAmount, blackTokensAmount);
```

with something, that is equally readable, for example:

```
return (
    _whiteToken.balanceOf(address(this)), _blackToken.balanceOf(address(this))
);
```

Optionally this optimization could be implemented in other contracts where possible, for example `Incentives.sol` (`governanceTokens`) or in `PrimaryCollateralizationERC20.sol`

### Re-audit:

Fixed.

# Unnecessary variables declaration

LOW | RESOLVED

Although the current code looks good, it can be a little bit more gas efficient. At contract CollateralizationERC20.sol there are 3 new variables:

```
IERC20 _whiteToken;
IERC20 _blackToken;
IERC20 _collateralToken;
```

But it isn't necessarily to do so, you can call token transfer directly:

```
IERC20(whiteTokenAddress).transfer(destination, tokensAmount)
```

This will save some gas on deployment + very small amount on every function call, here is brief test that shows gas cost of deployment and transaction of two different methods:

```

153     }
154   }
155
156   contract SendToken1 {
157
158     address public token1Address;
159     address public token2Address;
160
161     IERC20 token1;
162     IERC20 token2;
163
164     constructor (address _token1, address _token2){
165         token1Address = _token1;
166         token2Address = _token2;
167
168         token1 = IERC20(_token1);
169         token2 = IERC20(_token2);
170     }
171
172     function sendTokens() public {
173
174         for(uint256 c = 0; c < 30; c++){
175             token1.transfer(0xB2036E11eD0822304EE17A2d74B4652F8d7b89F7, 11111);
176             token2.transfer(0xB2036E11eD0822304EE17A2d74B4652F8d7b89F7, 22222);
177         }
178
179     }
180 }
181
182 }
183
184

```

deploy 374435

transaction 580862

```

185 contract SendToken2 {
186
187     address public token1Address;
188     address public token2Address;
189
190     constructor (address _token1, address _token2){
191         token1Address = _token1;
192         token2Address = _token2;
193     }
194
195     function sendTokens() public {
196
197         for(uint256 c = 0; c < 30; c++){
198             IERC20(token1Address).transfer(0xB2036E11eD0822304EE17A2d74B4652F8d7b89F7, 11111);
199             IERC20(token2Address).transfer(0xB2036E11eD0822304EE17A2d74B4652F8d7b89F7, 22222);
200         }
201
202     }
203
204 }
205
206
207

```

deploy 323517

transaction 580772

**Recommendation:**

Unless there is a special requirement, please replace pre stored variables with interface wrapper.

**Re-audit:**

Fixed.

## Unnecessary variables declaration

LOW | RESOLVED

At contract Reservoir.sol at function drip there are 2 new variable, that doesn't seem to have any sense: target\_ and token\_.

**Recommendation:**

Access target and token directly, remove:

```

address target_ = target;
IERC20 token_ = token;

```

**Re-audit:**

False finding. The original code is more gas efficient than the suggested one.



## Gas efficiency could be improved

LOW | RESOLVED

At contract `Incentives.sol`, `withdrawAllIncentives` function there is a loop that constantly checks `allTokens` array length:

```
for(uint256 i=0;i<allTokens.length;i++)
```

Also, `nonReentrant` protection in this case seems unnecessary, since function works only with known tokens that don't do any callback calls.

### **Recommendation:**

Pre store tokens length in a variable before cycle, this will be a little bit more gas efficient. And remove `nonReentrant` modifier.

### **Re-audit:**

Fixed, although there are still several other cases across project, for example in `Incentives.sol` contract in `userState`, `availableTokens` functions.

## Unnecessary variables declaration

LOW | RESOLVED

Although the current code looks very good, it can be a little bit more gas efficient. At contract `PrimaryPoolERC20.sol` there is a function `getBWprice()` that creates several variables.

We can try to do some optimization, to save gas fees. It is important to keep the function well readable as it is now.

### Recommendation:

Replace the function with, for example, this option:

```
function getBWprice() public view returns (uint256) {
    // Get data from collateralization contract
    uint256 curSupply = _collateralization.getWhiteSupply().add(_collateralization.getBlackSupply());
    /*
        Calculate token price. If current collateralization is less than default price use default price.
        If token collateralization is higher than default price define new price from collateralization.
    */
    if (curSupply == 0) {
        return _blackAndWhitePrice;
    }

    uint256 newPrice = wdiv(_collateralization.getCollateralization(), curSupply);
    return (newPrice > _blackAndWhitePrice) ? newPrice : _blackAndWhitePrice;
}
```

Also note spell fix at comment: `define`.

### Re-audit:

Fixed (in better way than suggested one, spell check skipped though).

## Gas efficiency could be improved

LOW | RESOLVED

At contract Incentives.sol, putIncentivesTokens function there is a loop that constantly checks allTokens array length:

```
for(uint256 i=0;i<allTokens.length;i++)
```

Also, this check doesn't seem to make sense:

```
if(amount<users[msg.sender].borrowedList[tokenAddress]){
  users[msg.sender].borrowedList[tokenAddress] =
  users[msg.sender].borrowedList[tokenAddress].sub(amount);
}else{
  users[msg.sender].borrowedList[tokenAddress] = 0;
}
```

Since the amount already been checked at line 130:

```
require(amount>0 && users[msg.sender].borrowedList[tokenAddress] >= amount,"bad amount or
tokenAddress");
```

So the else statement will always be false.

### Recommendation:

Pre store tokens length in a variable before cycle, this will be a little bit more gas efficient. And remove remove the if condition, keep only:

```
users[msg.sender].borrowedList[tokenAddress] =
users[msg.sender].borrowedList[tokenAddress].sub(amount);
```

### Re-audit:

Length - Fixed, else statement - Skipped.

## Unnecessary variables declaration

LOW | RESOLVED

At contract CollateralizationERC20.sol at function buy there are 2 new variable that used only once:

```
uint256 blackAndWhitePrice = getBWprice();  
uint256 oneTokenAmount = tokensAmount.div(2);
```

### Recommendation:

Unless code becomes hard to read, don't create new variable, simply insert function as argument into another function:

```
uint256 tokensAmount = wdiv(payment, getBWprice());  
_collateralization.buy(msg.sender, tokensAmount.div(2), payment);
```

### Re-audit:

Fixed.

## Unspecified revert reason

LOW | RESOLVED

At contract CollateralizationERC20.sol at function buy there is a condition:

```
require(tokensAmount >= _minBlackAndWhiteBuy);
```

If the transaction reverts it may be unclear why.

### Recommendation:

Add message with brief description:

```
require(tokensAmount >= _minBlackAndWhiteBuy, "tokens amount is less than minimum amount");
```

### Re-audit:

Fixed.

## Unexpected words order

INFORMATIONAL | RESOLVED

At 6 different contracts there are 39 occurrences of "...should be not..." expression, example: "WHITE TOKEN ADDRESS SHOULD BE NOT NULL".

I believe the correct word order is "...SHOULD NOT BE...".

### Recommendation:

Replace "...SHOULD BE NOT NULL" with "...SHOULD NOT BE NULL". And other 38 variation of this message.

### Re-audit:

Fixed.

## No SPDX-License-Identifier found in: Reservoir.sol, FarmingPool.sol

INFORMATIONAL | RESOLVED

Trust in smart contracts can be better established if their source code is available. Since making source code available always touches on legal problems with regards to copyright, the Solidity compiler encourages the use of machine-readable SPDX license identifiers.

### Recommendation:

Every source file should start with a comment indicating its license.

### Example:

```
// SPDX-License-Identifier: UNLICENSED
```

### Re-audit:

Fixed.

# Misleading formulas in the comments with the formulas in the function

INFORMATIONAL | RESOLVED

Contract Bmath.sol function calcSingleInGivenPoolOut:

```

*****
// calcSingleInGivenPoolOut
// tAi = tokenAmountIn
// pS = poolSupply
// pAo = poolAmountOut
// bI = balanceIn
// wI = weightIn
// tW = totalWeight
// sF = swapFee
*****

```

$$tAi = \frac{((pS + pAo) \cdot bI - bI \cdot (wI / tW)) \cdot sF}{(1 - (wI / tW)) \cdot sF}$$

Formula according to the code in the denominator:

```

*****
/      /      wI \      \
| 1 - | 1 - ---- | * sF |
\      \      tW /      /
*****


```

Contract Bmath.sol function calcPoolInGivenSingleOut:

```

*****
// calcPoolInGivenSingleOut
// pAi = poolAmountIn
// bO = tokenBalanceOut
// tAo = tokenAmountOut
// ps = poolSupply
// wO = tokenWeightOut
// tW = totalWeight
// sF = swapFee
// eF = exitFee
*****

```

$$pAi = \frac{bO \cdot (1 - ((1 - (tO / tW)) \cdot sF)) \cdot wO}{(1 - eF)}$$


### Recommendation:

Change comments according to the formula.Code Coverage and Test Results for all files.

### Re-audit:

Fixed.

# CODE COVERAGE AND TEST RESULTS FOR ALL FILES

## Tests written by Polars team

### Code Coverage

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	UNCOVERED LINES
contracts\	98.66	90.17	93.81	98.66	
BConst.sol	100.00	100.00	100.00	100.00	
BFactory.sol	100.00	75.00	100.00	100.00	
BMath.sol	90.91	100.00	53.85	90.91	... 233, 294, 358
BNum.sol	100.00	90.63	100.00	100.00	
BPool.sol	100.00	90.96	100.00	100.00	
BToken.sol	100.00	80.00	100.00	100.00	
<b>All files</b>	<b>98.66</b>	<b>90.17</b>	<b>93.81</b>	<b>98.66</b>	

### Test Results

**Contract: BPool**

Extreme weights

- ✓ swapExactAmountIn (1245ms)
- ✓ swapExactAmountOut (763ms)
- ✓ joinPool (2024ms)
- ✓ exitPool (898ms)
- ✓ joinswapExternAmountIn (1142ms)
- ✓ joinswapPoolAmountOut (1346ms)
- ✓ joinswapExternAmountIn should revert (1830ms)
- ✓ joinswapPoolAmountOut should revert (6194ms)
- ✓ exitswapExternAmountOut should revert (310ms)



- ✓ exitswapPoolAmountIn should revert (360ms)
- ✓ exitswapExternAmountOut (933ms)
- ✓ poolAmountOut = joinswapExternAmountIn(joinswapPoolAmountOut(poolAmountOut)) (913ms)
- ✓ tokenAmountIn = joinswapPoolAmountOut(joinswapExternAmountIn(tokenAmountIn)) (1098ms)
- ✓ poolAmountIn = exitswapExternAmountOut(exitswapPoolAmountIn(poolAmountIn)) (956ms)
- ✓ tokenAmountOut = exitswapPoolAmountIn(exitswapExternAmountOut(tokenAmountOut)) (989ms)

### Contract: BFactory

#### Factory

- ✓ isBPool on non pool returns false (130ms)
- ✓ isBPool on pool returns true (67ms)
- ✓ fails nonAdmin calls collect (199ms)
- ✓ admin collects fees (2582ms)
- ✓ nonadmin cant set admin address (180ms)
- ✓ admin changes admin address (208ms)

### Contract: BPool

#### With fees

- ✓ swapExactAmountIn (379ms)
- ✓ swapExactAmountOut (478ms)
- ✓ joinPool (1344ms)
- ✓ exitPool (958ms)
- ✓ joinswapExternAmountIn (1732ms)
- ✓ joinswapPoolAmountOut (1955ms)
- ✓ exitswapPoolAmountIn (1830ms)
- ✓ exitswapExternAmountOut (2375ms)
- ✓ pAo = joinswapExternAmountIn(joinswapPoolAmountOut(pAo)) (1210ms)
- ✓ tAi = joinswapPoolAmountOut(joinswapExternAmountIn(tAi)) (1949ms)
- ✓ pAi = exitswapExternAmountOut(exitswapPoolAmountIn(pAi)) (1246ms)
- ✓ tAo = exitswapPoolAmountIn(exitswapExternAmountOut(tAo)) (2038ms)

### Contract: TMath

#### BMath

- ✓ badd throws on overflow (415ms)
- ✓ bsub throws on underflow (111ms)
- ✓ bmul throws on overflow (87ms)
- ✓ bdiv throws on div by 0 (96ms)
- ✓ bpow throws on base outside range (710ms)

**Contract: BPool**

Binding Tokens

- ✓ Controller is msg.sender (451ms)
- ✓ Pool starts with no bound tokens (198ms)
- ✓ Fails binding tokens that are not approved (714ms)
- ✓ Admin approves tokens (1858ms)
- ✓ Fails binding weights and balances outside MIX MAX (3092ms)
- ✓ Fails finalizing pool without 2 tokens (628ms)
- ✓ Admin binds tokens (2369ms)
- ✓ Admin unbinds token (1732ms)
- ✓ Fails binding above MAX TOTAL WEIGHT (647ms)
- ✓ Fails rebinding token or unbinding random token (1111ms)
- ✓ Get current tokens (150ms)
- ✓ Fails getting final tokens before finalized (73ms)

Finalizing pool

- ✓ Fails when other users interact before finalizing (1616ms)
- ✓ Fails calling any swap before finalizing (1391ms)
- ✓ Fails calling any join exit swap before finalizing (1195ms)
- ✓ Only controller can setPublicSwap (594ms)
- ✓ Fails setting low liquidity providers fee (344ms)
- ✓ Fails setting high liquidity providers fee (309ms)
- ✓ Fails setting high governance fee (328ms)
- ✓ Fails setting high collateralization fee (331ms)
- ✓ Fails setting high swaps fees (968ms)
- ✓ Fails nonadmin sets fees or controller (1075ms)
- ✓ Fails nonadmin sets governance or collateralization wallets (575ms)
- ✓ Admin sets liquidity providers fee (380ms)
- ✓ Admin sets governance fee (366ms)
- ✓ Admin sets collateralization fee (430ms)
- ✓ Admin sets swaps fees (910ms)
- ✓ Admin sets governance or collateralization wallets (484ms)
- ✓ Fails nonadmin finalizes pool (213ms)
- ✓ Admin finalizes pool (495ms)
- ✓ Fails finalizing pool after finalized (430ms)
- ✓ Cant setPublicSwap, set fees when finalized (1153ms)
- ✓ Fails binding new token after finalized (461ms)
- ✓ Fails unbinding after finalized (324ms)
- ✓ Get final tokens (88ms)

User interactions

- ✓ Other users approve tokens (1326ms)
- ✓ User1 joins pool (716ms)
- ✓ Fails admin unbinding token after finalized and others joined (270ms)
- ✓ getSpotPriceSansFee and getSpotPrice (203ms)
- ✓ Fail swapExactAmountIn unbound or over min max ratios (541ms)
- ✓ swapExactAmountIn (978ms)
- ✓ swapExactAmountOut (484ms)
- ✓ Fails joins exits with limits (2710ms)
- ✓ Fails calling any swap on unbound token (2099ms)
- ✓ Fails calling weights, balances, spot prices on unbound token (733ms)

BToken interactions

- ✓ Token descriptors (445ms)
- ✓ Token allowances (1305ms)
- ✓ Token transfers (859ms)

**Contract: BPool**

Binding Tokens

- ✓ Admin approves tokens (2676ms)
- ✓ Admin binds tokens (5298ms)
- ✓ Fails binding more than 8 tokens (387ms)
- ✓ Rebind token at a smaller balance (1213ms)
- ✓ Fails gulp on unbound token (443ms)
- ✓ Pool can gulp tokens (1091ms)
- ✓ Fails swapExactAmountIn with limits (2744ms)
- ✓ Fails swapExactAmountOut with limits (3148ms)

94 passing (3m)

## Tests written by Zokyo Security team

As part of our work assisting Polars in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Truffle testing framework.

Tests were based on the functionality of the code, as well as a review of the Polars contract requirements for details about issuance amounts and how the system handles these.

### Code Coverage

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	UNCOVERED LINES
FarmingPool.sol	98.85	70.83	100.00	98.85	240
Reservoir.sol	100.00	50.00	100.00	100.00	
CollateralizationERC20.sol	100.00	94.23	100.00	100.00	
PrimaryCollateralizationERC20.sol	100.00	92.86	100.00	100.00	
PrimaryPoolERC20.sol	100.00	100.00	100.00	100.00	
BPool.sol	100.00	51.06	100.00	100.00	
VotingeScrow.sol	98.00	80.00	100.00	98.00	
BMath.sol	98.48	100.00	92.31	98.48	
BFactory.sol	100.00	75.00	100.00	100.00	
Incentives.sol	95.70	88.46	94.12	94.79	40, 41, 42 ,43, 165
<b>All files</b>	<b>99.07</b>	<b>80.03</b>	<b>98.08</b>	<b>99.07</b>	

### Test Results

```

Contract: FarmingPool
  On setup
    ✓ should deploy (11898ms)
  
```

On success

On pool info

- ✓ should call poolLength (108ms)

On invest

- ✓ should deposit to first pool [500000000] (1850ms)
- ✓ should call emergencyWithdraw (691ms)
- ✓ should deposit to second pool [500000000] (1203ms)
- ✓ should withdraw (1250ms)
- ✓ should withdraw [with rewards] (2341ms)
- ✓ should call getMultiplier (123ms)
- ✓ should deposit to first pool [2] (3819ms)
- ✓ should withdraw [2] (1966ms)
- ✓ should deposit [when block.timestamp <= pool.lastReward] (1351ms)
- ✓ should withdraw [when block.timestamp <= pool.lastReward] (648ms)

**Contract: BPool**

Setup

- ✓ should deploy (671ms)
- ✓ should get constant info about token (825ms)

On token

- ✓ should check balance of admin (93ms)
- ✓ should transfer tokens to user (133ms)
- ✓ should NOT tranfer [ERR\_INSUFFICIENT\_BAL] (910ms)
- ✓ should increase allowance [0.2 eth] and transfer [0.1 eth] (669ms)
- ✓ should call allowance (62ms)
- ✓ should decrease allowance and NOT transfer (986ms)

**Contract: BPool**

Setup

- ✓ should deploy (9854ms)

Set/Get

Fee

- ✓ LiquidityProviders (1442ms)
- ✓ Governance (988ms)
- ✓ Collateralization (318ms)
- ✓ SwapFee (1089ms)

Address

- ✓ GovernanceWallet (1446ms)

- ✓ Controller (313ms)
- ✓ CollateralizationWallet (867ms)

Other

- ✓ PublicSwap (328ms)

On bind

- ✓ should bind tokens (12724ms)
- ✓ isBound (628ms)
- ✓ should call getNumTokens (172ms)
- ✓ should call getCurrentTokens (112ms)
- ✓ should call getDenormalizedWeight (110ms)
- ✓ should call getTotalDenormalizedWeight (161ms)
- ✓ should call getNormalizedWeight (305ms)
- ✓ should call getBalance (143ms)
- ✓ should call getSpotPrice (154ms)
- ✓ should call getSpotPriceSansFee (245ms)
- ✓ should call gulp (607ms)
- ✓ should unbind token (855ms)
- ✓ should rebind token (450ms)

On finalized

- ✓ should call finilez (1229ms)
- ✓ should call getFinalTokens (127ms)
- ✓ should join pool (1035ms)
- ✓ should exit pool (851ms)

On swap

- ✓ should call swapExactAmountIn (1983ms)
- ✓ should call swapExactAmountOut (1097ms)
- ✓ should call joinswapExternAmountIn (1363ms)
- ✓ should call joinswapPoolAmountOut (744ms)
- ✓ should call exitswapPoolAmountIn (757ms)
- ✓ should call exitswapExternAmountOut (987ms)

**Contract: Collateralization**

On setup

- ✓ should deploy (3290ms)

On success

- ✓ should buy (4180ms)
- ✓ should buy [separate] (2130ms)
- ✓ should call buyBack (3516ms)

- ✓ should call buyBackSeparately (3345ms)
- ✓ should change pool address (654ms)
- ✓ should change governance address (301ms)
- ✓ should call getCollateralization (124ms)
- ✓ should call getStoredTokensAmount (565ms)

On Reverts

- ✓ should NOT create instanse [WHITE TOKEN zero address] (615ms)
- ✓ should NOT create instanse [BLACK TOKEN zero address] (767ms)
- ✓ should NOT create instanse [COLLATERAL TOKEN zero address] (1075ms)
- ✓ should NOT call func [caller is not pool] (750ms)
- ✓ should NOT call func [caller is not govern] (221ms)
- ✓ should NOT buy [DESTINATION ADDRESS SHOULD NOT BE NULL] (567ms)
- ✓ should NOT buy [NOT ENOUGH WHITE TOKENS ON COLLATERALIZATION CONTRACT BALANCE] (3221ms)
- ✓ should NOT buy [NOT ENOUGH BLACK TOKENS ON COLLATERALIZATION CONTRACT BALANCE] (4241ms)
- ✓ should NOT buy [NOT ENOUGH DELEGATED TOKENS] (4446ms)
- ✓ should NOT buy separate [DESTINATION ADDRESS SHOULD NOT BE NULL] (3469ms)
- ✓ should NOT buy separate [NOT ENOUGH DELEGATED TOKENS] (2903ms)
- ✓ should NOT buy separate [NOT ENOUGH WHITE TOKENS ON COLLATERALIZATION CONTRACT BALANCE] (3534ms)
- ✓ should NOT buy separate [NOT ENOUGH BLACK TOKENS ON COLLATERALIZATION CONTRACT BALANCE] (3854ms)
- ✓ should NOT buy back [DESTINATION ADDRESS SHOULD NOT BE NULL] (2209ms)
- ✓ should NOT buy back [NOT ENOUGH COLLATERALIZATION IN THE CONTRACT] (2003ms)
- ✓ should NOT buy back [NOT ENOUGH DELEGATED WHITE TOKENS ON DESTINATION BALANCE] (2713ms)
- ✓ should NOT buy back [NOT ENOUGH DELEGATED BLACK TOKENS ON DESTINATION BALANCE] (2798ms)
- ✓ should NOT buy back separate [DESTINATION ADDRESS SHOULD NOT BE NULL] (1917ms)
- ✓ should NOT buy back separate [NOT ENOUGH COLLATERALIZATION ON THE CONTRACT] (1751ms)
- ✓ should NOT buy back separate [NOT ENOUGH DELEGATED WHITE TOKENS ON DESTINATION BALANCE] (2127ms)
- ✓ should NOT buy back separate [NOT ENOUGH DELEGATED BLACK TOKENS ON DESTINATION BALANCE] (2473ms)

**Contract: PrimaryPoolIERC20**

On setup

- ✓ should deploy (6015ms)

On PrimaryPoolERC20

On buy

- ✓ should buy [1] (1319ms)
- ✓ should NOT buy [Buy amount is too low] (576ms)
- ✓ should change buy limit (398ms)
- ✓ should buy [2] (1356ms)
- ✓ should buy [3] (1623ms)
- ✓ should call buyBack (984ms)

On reverts

- ✓ should NOT change governanceAddress [zero addr] (381ms)
- ✓ should change governanceAddress (408ms)
- ✓ should NOT call func [sender not allowed] (360ms)
- ✓ should NOT create primaryPool [Collateralization address is zero] (443ms)
- ✓ should NOT call buyBack [Not enough collateral on the contract] (448ms)

On CollateralizationERC20

On reverts

- ✓ should NOT create new instance [Collateral token zero address] (530ms)
- ✓ should NOT call func [only pool can] (359ms)
- ✓ should NOT call func [only governance can] (382ms)
- ✓ should NOT buy [Destination SHOULD NOT BE NULL] (284ms)
- ✓ should NOT change pool address [zero addr] (293ms)
- ✓ should NOT change govern address [zero addr] (254ms)

On success

- ✓ should change govern address (473ms)
- ✓ should change bwt owner (378ms)
- ✓ should change pool address (444ms)

**Contract: BMath**

BMath Test Cases

- ✓ Should calculate spot price correctly (398ms)
- ✓ Should calculate the amount of tokens out from given in correctly (827ms)
- ✓ Should calculate the amount of tokens in from given out correctly (757ms)
- ✓ Should calculate pool amount out from pool shares correctly (1280ms)
- ✓ Should calculate pool amount in from pool shares correctly (2114ms)
- ✓ Should calculate token amount out correctly (2050ms)
- ✓ Should calculate pool amount in correctly (1236ms)



**Contract: Factory**

- ✓ should deploy with correct admin (162ms)
- ✓ should set new admin correctly (1142ms)
- ✓ shouldn't set new admin if caller is not current admin (1057ms)
- ✓ should create new BPool correctly (1496ms)
- ✓ should collect fees correctly (7762ms)

**Contract: BNum**

- ✓ Should throw on overflow (105ms)
- ✓ Should throw on underflow (99ms)
- ✓ Should throw on overflow (112ms)
- ✓ Should throw on division by 0 (97ms)
- ✓ Should throw on too low base (113ms)
- ✓ Should throw onto high base (109ms)

**Contract: VotingScrow**

Setup

- ✓ should deploy (383ms)

On deposit

- ✓ should NOT deposit [before first lock] (857ms)
- ✓ should create first lock (418ms)
- ✓ should NOT deposit [0 eth] (85ms)
- ✓ should deposit for (341ms)
- ✓ should withdraw (245ms)

On crowdsale

- ✓ should change controller (54ms)
- ✓ should set crowdsale (68ms)
- ✓ should create lock [for origin] (254ms)
- ✓ should increase lock amount and duration (341ms)
- ✓ should call balanceOf (84ms)
- ✓ should call balanceOfAt (139ms)
- ✓ should call totalSupplyAt (78ms)

On owners

- ✓ should NOT call func [without permission] (98ms)
- ✓ should NOT transfer ownership [not committed yet] (62ms)
- ✓ should transfer ownership (240ms)

**Contract: Incentives**

Setup

✓ should deploy (4524ms)

On success

- ✓ should create new incentive token (3837ms)
- ✓ should set crowdsale (539ms)
- ✓ should lock governance (1841ms)
- ✓ should put governance (3454ms)
- ✓ should lock governance (685ms)
- ✓ should withdraw governance (523ms)
- ✓ should withdraw incentives tokens (1628ms)
- ✓ should put incentives tokens (1136ms)
- ✓ should call userState (174ms)
- ✓ should call availableTokens (477ms)
- ✓ should call userscount (89ms)
- ✓ should call getMaxBorrowed (76ms)
- ✓ should call getGovBalances (116ms)
- ✓ should call getAddress (289ms)
- ✓ should switch owner (287ms)

On reverts

- ✓ should NOT call func [not owner] (903ms)
- ✓ should NOT create new incentive token (534ms)
- ✓ should NOT put governance (519ms)
- ✓ should NOT lock governance (758ms)
- ✓ should NOT withdraw (1551ms)
- ✓ should NOT call putIncentivesTokens (883ms)
- ✓ should NOT call getMaxBorrowed (121ms)
- ✓ should NOT call getGovBalances (134ms)
- ✓ should NOT call userState (101ms)

162 passing (4m 27s)

We are grateful to have been given the opportunity to work with the Polars team.

**The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.**

Zokyo's Security Team recommends that the Polars team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

**ZOKYO.**