Security Assessment Report

Poly-notes

February 28, 2023

Summary

The sec3 team (formerly Soteria) was engaged to do a thorough security analysis of the Poly-notes smart contract program. The artifact of the audit was the source code of the following on-chain smart contract excluding tests in a private repository.

The audit was done on the following contract

- **Contract** "poly-notes"**:**
    - Tag Mainnet-Alpha-0.1.0
    - Commit d5e61c91905a6bb164658b652b650ba2fcd2ee86

The audit revealed 8 issues or questions. The team responded with the following commits for a post-audit review, which is to confirm if the reported issues have been resolved.

    - PRs #51, #52, #53, #54, #55, #56, #57, #62, #64

This report describes the findings and resolutions in detail.

Table of Contents

Methodology and Scope of Work

The sec3 (formerly Soteria) audit team, which consists of Computer Science professors and industrial researchers with extensive experience in smart contract security, program analysis, testing and formal verification, performed a comprehensive manual code review, software static analysis and penetration testing.

Assisted by the sec3 Scanner developed in-house, the audit team particularly focused on the following work items:

- Check common security issues.
    - Reentrancy
    - Unchecked call return value
    - Integer overflow/underflow
    - Address hardcoded
    - Missing zero address validation
    - Delegatecall to untrusted call
    - Dos with failed call
    - Presence of unused variables
    - Dos with block gas limit
    - Timestamp dependence
    - Arithmetic accuracy
    - Outdated dependencies
    - Redundant code
    - Cross-function race conditions
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of scope of this work.

Result Overview

In total, the audit team found the following issues.

CONTRACT POLY-NOTES MAINNET-ALPHA-0.1.0

Issue	Impact	Status
[C-1] Unsafe function overriding and unchecked call return value	Critical	Resolved
[C-2] Invoice without paying	Critical	Resolved
[M-1] Missing zero address check	Medium	Resolved
[L-1] The issuer and operator should not be the same	Low	Resolved
[I-1] Local variable not initialized	Informational	Resolved
[I-2] Missing maturityDate constraints	Informational	Resolved
[I-3] Inconsistent comment in BulletBond test	Informational	Resolved
[I-4] Missing unitPrice checks	Informational	Resolved

Findings in Detail

IMPACT – CRITICAL

[C-1] Unsafe function overriding and unchecked call return value

1. Unsafe function overriding

```
/* contracts/BulletBond/BondData.sol */
024 | contract BondData {
031 |     IERC20 public currency;
```

If an untrusted ERC20 contract address is passed as the currency parameter and that contract has improperly overridden the transfer or transferFrom functions, the untrusted contract can lead to vulnerabilities such as the reentrancy and the damages caused by the unchecked return value described below.

2. Bypass payment

If the transferFrom or transfer function returns false, indicating that the transfer has failed, the code after the transfer will still execute.

```
/* contracts/BulletBond/BulletBond.sol */
099 | function _repay() internal {
100 |   require(issuanceConcluded, 'BulletBond: Issuance not concluded.');
101 |   require(!issuerPaid, 'BulletBond: Already repaid.');
102 |
103 |   issuerPaid = true;
104 |   bondData.currency().transferFrom(_msgSender(), address(this), _totalSettlementAmount());
105 |       emit Repay(_msgSender(), _totalSettlementAmount());
106 |    }

/* contracts/BulletBond/BulletBond.sol */
112 | function _settleNotes(uint256 _amount) internal {
113 |     require(block.timestamp > bondData.maturityDate(), 'BulletBond: Maturity date not reached');
114 |     require(issuerPaid, 'BulletBond: Issuer has not paid yet.');
115 |
116 |     _burn(_msgSender(), _amount);
117 |     bondData.currency().transfer(_msgSender(), _amount * _unitSettlementAmount());
118 |     emit SettleNotes(_msgSender(), _amount);
119 | }
```

```
/* contracts/BulletBond/IssuanceProgram.sol */
202 | function _fundOrder() internal {
203 |     uint256 _allocatedUnits = allocatedUnits[_msgSender()];
204 |     require(_allocatedUnits > 0, 'IssuanceProgram: Account has no allocated units');
205 |
206 |     uint256 amount = _allocatedUnits - fundedUnits[_msgSender()];
207 |     require(amount > 0, 'IssuanceProgram: Order already funded');
208 |
209 |     unchecked {
210 |         fundedUnits[_msgSender()] += amount;
211 |     }
212 |
213 |     bondData.currency().transferFrom(_msgSender(), address(this), amount * unitPrice);

/* contracts/BulletBond/IssuanceProgram.sol */
286 | function _redeemIssuanceProceeds() internal {
287 |     uint256 issuanceProceeds = bondData.currency().balanceOf(address(this));
288 |     bondData.currency().transfer(issuerPaymentAddress, issuanceProceeds);
289 |
290 |     if (fundingConcluded) {
291 |         _concludeIssuance();
292 |     }
293 |
294 |     emit RedeemIssuanceProceeds(_msgSender(), issuerPaymentAddress, issuanceProceeds);
295 | }

/* contracts/Invoice.sol */
028 | function pay() public notPaid {
029 |     paid = true;
030 |     currency.transferFrom(_msgSender(), recipient, amount);
031 |     emit Pay(_msgSender(), amount);
032 | }
```

PoC

We directly return `false` in `transferFrom` and `transfer` function in `contracts/Mintable.sol`.

```
pragma solidity ^0.8.0;

import '@openzeppelin/contracts/token/ERC20/ERC20.sol';
import '@openzeppelin/contracts/utils/Context.sol';
```

```
contract Mintable is ERC20 {
    constructor(string memory name, string memory symbol) ERC20(name, symbol) {}

    function mint(address account, uint256 amount) public {
        _mint(account, amount);
    }

    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public virtual override returns (bool) {
        return false;
    }

    function transfer(address to, uint256 amount) public virtual override returns (bool) {
        address owner = _msgSender();
        _transfer(owner, to, amount);
        return false;
    }
}
```

Add a test case in `test/BulletBond.ts` in line `117`:

```
    it('attack repay', async function () {
      const { bulletBond, config, operator, issuer, investor, testCurrency } = await loadFixture(
        deployBulletBond
      );
      const { amountPurchased, coupon, principal } = config;

      await bulletBond.mint(investor.address, amountPurchased);

      await bulletBond.connect(operator).concludeIssuance();

      bulletBond.connect(issuer).repay()
      bulletBond.connect(issuer).repay()
      bulletBond.connect(issuer).repay()
    });
```

The test case will pass.

Potential repairs

1. Unsafe function overriding

If the contract chooses to use externally rewritten `transfer` or `transferFrom` functions, it is imperative to ensure that the `currency` contract has been thoroughly audited and can be fully trusted to prevent potential vulnerabilities.

2. Unchecked return value

Check the return value for every `transfer` or `transferFrom` functions like below:

```
require(bondData.currency().transferFrom(_msgSender(), address(this), _totalSettlementAmount()),
        "Token transfer failed!");
```

Resolution

This issue was fixed in PR #51.

The team understood the attack vectors via unsafe function overriding. The team stated that only USDC and other trusted ERC20 tokens will be used as currency options.

IMPACT – CRITICAL

[C-2] Invoice without paying

Once the `issueDate` has been reached, the issuer has the ability to bypass the payment to invoice by calling the `fundOrder` function.

After the `issueDate` has been reached, the fundholder can mint their BulletBond token based on their prior investments by calling the issue function. Meanwhile, the fundOrder function will automatically call the `_issue` function and also execute `_redeemIssuanceProceeds` to transfer funds to the issuer on behalf of the fundholder.

```
/* contracts/BulletBond/IssuanceProgram.sol */
225 | // In case of late funding we issue immediately and the issuer also receives the proceeds immediately.
226 | // RedeemIssuanceProceeds would also conclude the issuance if above fundingConcluded condition is set to true.
227 | if (_issuanceDateReached()) {
228 |     _issue(_msgSender());
229 |     _redeemIssuanceProceeds();
230 | }
```

Normally, the `invoiceAmount` is expected to be paid to the operator before transferring funds to the issuer. However, if a user calls the `fundOrder` function after the `issuanceDate` but before the `redeemIssuanceProceeds` function is executed, they may bypass the portion of the `redeemIssuanceProceeds` function responsible for paying the invoice.

```
/* contracts/BulletBond/IssuanceProgram.sol */
297 | function redeemIssuanceProceeds() public whenIssuanceDateReached noReentrant whenNotPaused {
298 |     if (!invoice.paid()) {
299 |         invoice.currency().approve(address(invoice), invoice.amount());
300 |         invoice.pay();
301 |     }
302 |     _redeemIssuanceProceeds();
303 | }
```

PoC

This test is failing because the invoice fee cannot be paid.

```
    it('attack invoice without paying', async function () {
      // Set date to be on the open date
```

```
    await time.increaseTo(issuanceDate);

    // Investor funds his order
    await expect(issuanceProgram.connect(investor).fundOrder()).to.changeTokenBalance(
      testCurrency,
      investor,
      -amountPurchased * principal
    );

    // Issuer redeems Issuance proceeds and pays the fees to the operator
    await expect(issuanceProgram.connect(issuer).redeemIssuanceProceeds())
      .to.changeTokenBalance(testCurrency, issuer, amountPurchased * principal - feesAmount)
      .to.changeTokenBalance(testCurrency, operator, feesAmount);
  });
```

Potential repairs

Use the `redeemIssuanceProceeds` function in the `fundOrder` function.

Resolution

This issue was fixed in PR #52 and #64.

IMPACT – MEDIUM

[M-1] Missing zero address checking

```
/* contracts/Invoice.sol */
21    constructor(IERC20 _currency, uint256 _amount, address _recipient) {
22        currency = _currency;
23        amount = _amount;
24        recipient = _recipient;
25        _transferOwnership(_recipient);
26    }

/* contracts/BulletBond/IssuanceProgram.sol */
191   function updateIssuerPaymentAddress(
192       address _issuerPaymentAddress
193   ) public onlyIssuer whenNotPaused {
194       _updateIssuerPaymentAddress(_issuerPaymentAddress);
195   }
```

The code above fails to verify whether the recipient address is zero, and it also lacks checks for the _transferOwnership function.

Similarly, _issuerPaymentAddress in line 194 should not be zero too.

Potential repairs

```
    constructor(IERC20 _currency, uint256 _amount, address _recipient) {
        currency = _currency;
        amount = _amount;
+       require(recipient != address(0));
        recipient = _recipient;
        _transferOwnership(_recipient);
    }
```

Resolution

This issue was fixed in PR #53 and #64.

IMPACT – LOW

[L-1] The issuer and operator should not be the same

When initializing a BulletBond contract, it should verify that the issuerSigningAddress is not the same as the operator address. If they are the same, the approveVoidance function will not work properly.

To ensure proper functionality of the voidance feature in the BulletBond contract, the issuer and operator must have distinct addresses. If they share the same address, the requirement that _msgSender() != request.requester would never be met, rendering the two voidance functions unuseful.

PoC

```
    it('same issuer: request and approve voidance', async function () {
        const { bulletBond, config, operator, investor, investor2 } = await loadFixture(
          deployBulletBond
        );
        // issuer and operator are the same
        const issuer = operator;
        const { amountPurchased } = config;
        // operator issues the bonds
        await bulletBond.mint(investor.address, amountPurchased);
        // check that bonds were minted to investor
        expect(await bulletBond.balanceOf(investor.address)).to.equal(amountPurchased);
        // Operator requests voidance
        await bulletBond.requestVoidance(1, investor.address, investor2.address, amountPurchased);
        // Issuer approves voidance
        await expect(
          bulletBond.
            connect(issuer).
            approveVoidance(1, investor.address, investor2.address, amountPurchased)
        ).to.be.rejectedWith('BulletBond: Requester cannot approve');
        // check that bonds are still with investor
        expect(await bulletBond.balanceOf(investor.address)).to.equal(amountPurchased);
      });
```

Potential repairs

```
/* contracts/BulletBond/BulletBond.sol */
 constructor(
     BondData _bondData,
     address _mintAuthority
 )
     Operable()
     WhitelistProtected(_bondData.whitelist())
     ERC20(_bondData.name(), _bondData.symbol())
 {
     require(_mintAuthority != address(0), 'BulletBond: mintAuthority is the zero address');
     require(_bondData.operator() != address(0), 'BulletBond: operator is the zero address');
+    require(_bondData.operator() != _bondData.issuerSigningAddress(), 'BulletBond: operator cannot
be the same as issuer');
     mintAuthority = _mintAuthority;
     bondData = _bondData;

     _transferOperatorAuthority(bondData.operator());
 }
```

Resolution

This issue was fixed in PR #54.

IMPACT – INFO

[I-1] Local variable not initialized

```
/* contracts/BulletBond/BulletBond.sol */
127 | function _requestVoidance(
132 | ) internal _onlyOperatorOrIssuer {
139 |     VoidanceRequest memory request;
140 |     request.amount = _amount;
141 |     request.holderAddress = _holder;
142 |     request.holderNewAddress = _newAddress;
143 |     request.requester = _msgSender();
144 |     request.approved = false;
145 |     voidanceRequests[_id] = request;
146 |     emit RequestVoidance(_id, _msgSender(), _holder, _newAddress, _amount);
147 | }
```

The above code does not initialize request.approvalDate.

Potential repairs

```
    function _requestVoidance(
...
        VoidanceRequest memory request;
        request.amount = _amount;
        request.holderAddress = _holder;
        request.holderNewAddress = _newAddress;
        request.requester = _msgSender();
        request.approved = false;
+       request.approvalDate = block.timestamp;
        voidanceRequests[_id] = request;
        emit RequestVoidance(_id, _msgSender(), _holder, _newAddress, _amount);
    }
```

Resolution

Fixed in PR #55.

IMPACT – INFO

[I-2] Missing maturityDate constraints

When initializing a BulletBond contract, it is recommended to constrain maturityDate to avoid incorrect configurations that may allow investors to settle immediately after the issuer repays or cause investors to be unable to settle at all.

Once the operator has completed the issuance and the issuer has repaid, investors are able to settle the bullet bond, provided that it has reached maturity. The maturityDate parameter is used to determine whether the bullet bond has matured.

If this parameter is set to a date prior to the conclusion or repayment time, investors may be able to settle the bond as soon as it is concluded and repaid.

If the maturityDate is set too far in the future, investors may never be able to settle the bond within a reasonable timeframe.

```
/* contracts/BulletBond/BulletBond.sol */
112 | function _settleNotes(uint256 _amount) internal {
113 |     require(block.timestamp > bondData.maturityDate(), 'BulletBond: Maturity date not
reached');
114 |     require(issuerPaid, 'BulletBond: Issuer has not paid yet.');
115 |
116 |     _burn(_msgSender(), _amount);
117 |     bondData.currency().transfer(_msgSender(), _amount * _unitSettlementAmount());
118 |     emit SettleNotes(_msgSender(), _amount);
119 | }
```

PoC

```
import { loadFixture, time } from '@nomicfoundation/hardhat-network-helpers';
import { expect } from 'chai';
import { ethers } from 'hardhat';
import { formatBytes32String } from 'ethers/lib/utils';

describe('Proof of Concept', async function () {
  async function deployBulletBond() {
    // Generate wallet addresses
    const [operator, issuer, investor, investor2, nonWhiteListedAddress] =
      await ethers.getSigners();

    // Deploy currency
```

```
const TestCurrency = await ethers.getContractFactory('Mintable');
const testCurrency = await TestCurrency.deploy('Test Coin', 'TEST');

// Time variables
const now = await time.latest();
const THREE_MONTHS_IN_SECS = 3 * 30 * 24 * 60 * 60;
const DAY_IN_SECS = 24 * 60 * 60;

// Deploy documents
const Documents = await ethers.getContractFactory('Documents');
const documents = await Documents.connect(issuer).deploy();
await documents.deployed();

// whitelist addresses
const Whitelist = await ethers.getContractFactory('Whitelist');
const whitelist = await Whitelist.connect(operator).deploy();
await whitelist.deployed();

await whitelist.connect(operator).setWhitelist(issuer.address, true);
await whitelist.connect(operator).setWhitelist(investor.address, true);
await whitelist.connect(operator).setWhitelist(operator.address, true);
await whitelist.connect(operator).setWhitelist(investor2.address, true);

const config = {
  principal: 1000,
  coupon: 100,
  amountPurchased: 10,
  initialBalance: 100000,
  maturityDate: now,            // make the maturity date earlier
};

// Mint 10000 test currency to the investor and the operator
await testCurrency.mint(investor.address, config.initialBalance);
await testCurrency.mint(issuer.address, config.initialBalance);
await testCurrency.mint(investor2.address, config.initialBalance);
await testCurrency.mint(nonWhiteListedAddress.address, config.initialBalance);

const bondDetails = {
  issuerSigningAddress: issuer.address,
  issuerName: 'test',
  operator: operator.address,
  principal: config.principal,
  coupon: config.coupon,
  currency: testCurrency.address,
  maturityDate: config.maturityDate,
  documents: documents.address,
  securityIdentifier: formatBytes32String(''),
  whitelist: whitelist.address,
```

```
    whitelistedTransfer: false,
    symbol: 'test',
    name: 'BulletBond',
  };

  const BondData = await ethers.getContractFactory('BondData');
  const bondData = await BondData.connect(operator).deploy(bondDetails);
  await bondData.deployed();

  const BulletBond = await ethers.getContractFactory('BulletBond');
  const bulletBond = await BulletBond.connect(operator).deploy(
    bondData.address,
    operator.address
  );
  await bulletBond.deployed();

  return {
    bulletBond,
    config,
    operator,
    issuer,
    investor,
    investor2,
    now,
    DAY_IN_SECS,
    THREE_MONTHS_IN_SECS,
    testCurrency,
    documents,
    whitelist,
    nonWhiteListedAddress,
  };
}

describe('Settle', function () {
  it('investors could settle immediately', async function () {
    const { bulletBond, config, operator, issuer, investor, testCurrency } = await loadFixture(
      deployBulletBond
    );
    const { amountPurchased, coupon, principal } = config;

    await bulletBond.mint(investor.address, amountPurchased);

    await bulletBond.connect(operator).concludeIssuance();

    // Issuer repays the bullet bond
    await testCurrency
      .connect(issuer)
      .approve(bulletBond.address, (principal + coupon) * amountPurchased);
```

```
      await expect(bulletBond.connect(issuer).repay()).to.changeTokenBalance(
        testCurrency,
        issuer,
        -amountPurchased * (principal + coupon)
      );

    //   await time.increaseTo(config.maturityDate);
      // Investor can settle immediately after conclusion
      await expect(bulletBond.connect(investor).settleNotes(amountPurchased)).to.changeTokenBalance(
        bulletBond,
        investor,
        -amountPurchased
      );
    });
  })
});
```

Resolution

Fixed in PR #57.

IMPACT – INFO

[I-3] Inconsistent comment in BulletBond test

```
/* test/BulletBond.ts */
717 | // Issuer requests voidance
718 | await bulletBond
719 |   .connect(issuer)
720 |   .requestVoidance(1, investor.address, investor2.address, amountPurchased);
721 |
722 | // Issuer approves voidance
723 | await bulletBond
724 |   .connect(operator)
725 |   .approveVoidance(1, investor.address, investor2.address, amountPurchased);
```

On line 722, it should be `Operator approves voidance`.

Resolution

Fixed in PR #56.

IMPACT – INFO

[I-4] Missing unitPrice checks

The contract should verify that the `unitPrice` value in the `IssuanceProgramDetails` contract matches the `principal` value in the `BondData` contract. Failure to do so could prevent investors from recovering the funds they invested.

The number of `bondData.currency()` invested by the user and the corresponding bulletBond token obtained is `unitPrice`.

```
/* contracts/BulletBond/IssuanceProgram.sol */
214 | bondData.currency().transferFrom(_msgSender(), address(this), amount * unitPrice);
215 |
216 | // mint bullet bond to issuance program to have accurate supply
217 | bulletBond.mint(address(this), amount);
```

When a user burns the `bulletBond` token and redeems the `bondData.currency()` asset, if `bondData.principal + bondData.coupon < unitPrice` , users will not receive profits and may even suffer losses.

The contract should make sure that `bondData.principal == unitPrice` so that the user's principal is not lost.

```
/* contracts/BulletBond/BulletBond.sol */
112 | function _settleNotes(uint256 _amount) internal {
113 |   require(block.timestamp > bondData.maturityDate(), 'BulletBond: Maturity date not reached');
114 |   require(issuerPaid, 'BulletBond: Issuer has not paid yet.');
115 |
116 |   _burn(_msgSender(), _amount);
117 |   bondData.currency().transfer(_msgSender(), _amount * _unitSettlementAmount());
118 |   emit SettleNotes(_msgSender(), _amount);
119 | }
```

PoC

```
it('attack lack of inspection', async function () {
    // Investor approves currency to be spent by bullet bond
    await testCurrency
        .connect(investor)
        .approve(issuanceProgram.address, amountPurchased * unitPrice);
```

```
    // Set date to be on the open date
    await time.increaseTo(issuanceDate);
    // Investor funds his order
    await expect(issuanceProgram.connect(investor).fundOrder()).to.changeTokenBalance(
          testCurrency,
          investor,
          -amountPurchased * unitPrice
    );
    // Issuer settles the bullet bond
    await testCurrency.connect(issuer)
          .approve(bulletBond.address, (principal + coupon) * amountPurchased);
    // issuer repay
    await expect(bulletBond.connect(issuer).repay()).to.changeTokenBalance(
          testCurrency,
          issuer,
          -amountPurchased * (principal + coupon)
    );
    await time.increaseTo(config.maturityDate);
    await expect(bulletBond.connect(investor).settleNotes(amountPurchased)).to.changeTokenBalance(
          testCurrency,
          investor,
          amountPurchased * (principal + coupon)
    );
    // get less than deposit
    expect(await testCurrency.balanceOf(investor.address)).to.lt(amountPurchased * unitPrice);
});
```

Potential repairs

```
    constructor(
        IssuanceProgramDetails memory _issuanceProgramDetails,
        BulletBondDetails memory _bondDetails
    ) Operable() WhitelistProtected(_bondDetails.whitelist) {
      ...
+     require(unitPrice == _bondDetails.principal, "Error Message");
      ...
    }
```

Resolution

Fixed in PR #62.

DISCLAIMER

The instance report ("Report") was prepared pursuant to an agreement between Coderrect Inc. d/b/a sec3 (the "Company") and FQX AG (the "Client"). This Report solely includes the results of a technical assessment of a specific build and/or version of the Client's code specified in the Report ("Assessed Code") by the Company. The sole purpose of the Report is to provide the Client with the results of the technical assessment of the Assessed Code. The Report does not apply to any other version and/or build of the Assessed Code. Regardless of the contents of the Report, the Report does not (and should not be interpreted to) provide any warranty, representation or covenant that the Assessed Code: (i) is error and/or bug free, (ii) has no security vulnerabilities, and/or (iii) does not infringe any third-party rights. Moreover, the Report is not, and should not be considered, an endorsement by the Company of the Assessed Code and/or of the Client. Finally, the Report should not be considered investment advice or a recommendation to invest in the Assessed Code and/or the Client.

This Report is considered null and void if the Report (or any portion thereof) is altered in any manner.

ABOUT

Founded by leading academics in the field of software security and senior industrial veterans, sec3 (formerly Soteria) is a leading blockchain security company. We are also building sophisticated security tools that incorporate static analysis, penetration testing, and formal verification.

At sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.