Porter Finance Security Review

Auditors

Brock Elmore, Lead Security Researcher

Satyam Agrawal, Security Researcher

DefSec, Security Researcher

Grmpyninja , Apprentice

Blockdev, Apprentice

Report prepared by: Pablo Misirov, Blockdev & Grmpyninja

May 27, 2022

Contents

1 About Spearbit **2**

2 Introduction **2**

3 Risk classification **2**
3.1 Impact . . . 2
3.2 Likelihood . . . 2
3.3 Action required for severity levels . . . 3

4 Executive Summary **3**

5 Findings **4**
5.1 Medium Risk . . . 4
5.1.1 Freeze Redeems if `bonds` too Large . . . 4
5.1.2 Reentrancy in `withdrawExcessCollateral()` and `withdrawExcessPayment()` functions. . . . 4
5.2 Low Risk . . . 5
5.2.1 `burn()` and `burnFrom()` allow users to lose their bonds . . . 5
5.2.2 Missing two-step transfer ownership pattern . . . 6
5.2.3 Inefficient initialization of minimal proxy implementation . . . 6
5.3 Gas Optimization . . . 7
5.3.1 Verify amount is greater than 0 to avoid unnecessarily `safeTransfer()` calls . . . 7
5.4 Informational . . . 8
5.4.1 Improve checks for token allow-list . . . 8
5.4.2 Incorrect revert message . . . 8
5.4.3 Non-existent bonds naming/symbol restrictions . . . 9
5.4.4 Needles variable initialization for default values . . . 9
5.4.5 Deflationary payment tokens are not handled in the `pay()` function . . . 10

6 Additional Comments **11**

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Porter's mission is to enable DAOs growth by providing them with access to credit.

Porter allows DAOs and other on-chain entities to create so-called "DeFi bonds", allowing borrowing capital when bonds are sold to lenders. The team is using smart contracts as agreements between borrowers and lenders similar to traditional zero coupon bonds, which can be sold at a discount on the market. Borrowers select parameters when they create their bond including a pair of tokens representing collateral and payment, conversion ratio, maturity and the number of bonds. As a result, bonds (based on ERC20 tokens) are minted and sold on the market to bondholders (lenders). Initially the bond contract only holds collateral supplied by the borrower (e.g. DAO), but when maturity is reached or the bond is fully repaid using a payment token, bondholders can redeem their assets.

The protocol does not implement any liquidation mechanisms and is purely based on the ratio between the collateral and payment tokens, meaning that it exposes lenders to market volatility.

This security review had a focus on, but was not limited to:

- Possibilities to steal or lock collateral or payment in bonds.
- Forced state transitions or state corruption.
- Incorrect arithmetic.
- Other manipulations and security concerns and risks for users and the protocol itself.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Porter Finance according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized

- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 10 days in total, Porter Finance engaged with Spearbit to review v1-core. In this period of time a total of 11 issues were found.

Summary

Project Name	Porter Finance
Repository	v1-core
Commit	985fb85f03917cb48...
Type of Project	Bonds, DeFi
Audit Timeline	May 2nd - May 11th
Methods	Manual Review, Foundry fuzzing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	2
Low Risk	3
Gas Optimizations	1
Informational	5
Total Issues	11

5 Findings

5.1 Medium Risk

5.1.1 Freeze Redeems if `bonds` too Large

Severity: *Medium Risk*

Context: Bond.sol#L309

Description: Issuing too many bonds can result in users being unable to redeem. This is caused by arithmetic overflow in `previewRedeemAtMaturity`.

If a user's `bonds` and`paidAmount`'s (or `bonds * nonPaidAmount`) product is greater than `2**256`, it will overflow, reverting all attempts to redeem bonds.

Recommendation: Implement a safety check in the factory as follows:

```
uint256 _safetyCheck_ = bonds * bonds;
```

Or inside the initialize function:

```
uint256 _safetyCheck_ = maxSupply * maxSupply;
```

Or change `bonds` in factory/`Bond.initialize` to a type of `uint128`.

This ensures that `bonds.mulDivDown(paidAmount, bondSupply)` is always computable without overflow, as `paidAmount` is at most `maxSupply` (set during `initialize`) and `bonds` is at most `maxSupply` (set during `initialize`). `bonds * bonds` passing the safety check ensures redeems remain functional.

Porter: Implemented in PR #290.

Spearbit: Acknowledged, recommendation has been implemented.

5.1.2 Reentrancy in `withdrawExcessCollateral()` and `withdrawExcessPayment()` functions.

Severity: *Medium Risk*

Context: Bond.sol#L212, Bond.sol#233

Description: `withdrawExcessCollateral()` and `withdrawExcessPayment()` enable the caller to withdraw excess collateral and payment tokens respectively. Both functions are guarded by an `onlyOwner` modifier, limiting their access to the owner of the contract.

```
function withdrawExcessCollateral(uint256 amount, address receiver) external onlyOwner
function withdrawExcessPayment(address receiver) external onlyOwner
```

When transferring tokens, execution flow is handed over to the token contract. Therefore, if a malicious token manages to call the owner's address it can also call these functions again to withdraw more tokens than required.

As an example consider the following case where the collateral token's `transferFrom()` function calls the owner's address:

```
function transferFrom(
    address from,
    address to,
    uint256 amount
) public virtual override returns (bool) {
    if (reenter) {
        reenter = false;
        owner.attack(bond, amount);
    }
    address spender = _msgSender();
    _spendAllowance(from, spender, amount);
    _transfer(from, to, amount);
    return true;
}
```

and the owner contract has a function:

```
function attack(address _bond, uint256 _amount) external {
    IBond(_bond).withdrawExcessCollateral(_amount, address(this));
}
```

When `withdrawExcessCollateral()` is called by `owner`, it allows it to withdraw double the amount via reentrancy.

Recommendation: Consider applying the `nonReentrant` modifier on both of these functions.

Porter: Implemented in PR #284.

Spearbit: Acknowledged, recommendation has been implemented.

5.2 Low Risk

5.2.1 `burn()` and `burnFrom()` allow users to lose their bonds

Severity: *Low Risk*

Context: Bond#L31

Description: The `Bond` contract inherits from 'ERC20BurnableUpgradeable.

```
contract Bond is
    IBond,
    OwnableUpgradeable,
    ERC20BurnableUpgradeable,
```

This exposes the `burn()` and `burnFrom()` functions to users who could get their bonds burned due to an error or a front-end attack.

Recommendation: Consider:

- Implementing an `onlyOwner` guard inside `burn()` because only the owner needs to be able to burn their bonds if these are not bought by any lenders.
- Any restriction to the `burnFrom(address account, uint256 amount)` function which has to be considered in context with the refinancing plan.

Porter: Due to low risk attack vector and additional complexity, we've decided to not restrict `burn()` and `burnFrom()` functions.

Spearbit: Acknowledged, recommendations have not been implemented.

5.2.2 Missing two-step transfer ownership pattern

Severity: *Low Risk*

Context: Bond.sol#L28

Description: After a bond is created its ownership is transferred to the wallet which invoked the `createBond` function, but it can be later transferred to anyone at any time or the `renounceOwnership` function can be called.

The `Bond` contract uses the `Ownable` Openzeppelin contract, which is a simple mechanism to transfer ownership without supporting a two-step ownership transfer pattern. OpenZeppelin describes `Ownable` as:

> Ownable is a simpler mechanism with a single owner "role" that can be assigned to a single account. This simpler mechanism can be useful for quick tests but projects with production concerns are likely to outgrow it.

Ownership transfer is a critical operation and transferring it to an inaccessible wallet or renouncing ownership by mistake can effectively lock the collateral in the contract forever.

Recommendation: It is recommended to implement a two-step transfer ownership mechanism where ownership is transferred and later claimed by a new owner to confirm the whole process and prevent a lockout.

Because the OpenZeppelin ecosystem does not provide such implementation it has to be developed in-house. For inspiration `BoringOwnable` can be considered. However, it has to be well tested especially in case it is integrated with other OpenZeppelin contracts used by the project.

Porter: Confirms and accepts the risks. The team will consider two-step ownership transfer for future contracts.

Spearbit: Acknowledged.

5.2.3 Inefficient initialization of minimal proxy implementation

Severity: *Low Risk*

Context: BondFactory.sol#L71, deploy_bond_factory.ts#L24

Description: The `Bond` contract uses a minimal proxy pattern when deployed by `BondFactory`. The proxy pattern requires a special `initialize` method to be called to set the state of each cloned contract.

Nevertheless, the implementation contract can be left uninitialized, giving an attacker the opportunity to invoke the initialization.

```
constructor() {
    tokenImplementation = address(new Bond());
    _grantRole(DEFAULT_ADMIN_ROLE, _msgSender());
}
```

After the reporting the issue it was discovered that a separate (not merged) development branch implements a deployment script which initializes the `Bond` implementation contract after the main deployment of `BondFactory`, leaving a narrow window for the attacker to leverage this issue and reducing impact significantly.

deploy_bond_factory.ts#L24

```
const implementationContract = (await ethers.getContractAt(
  "Bond",
  await factory.tokenImplementation()
)) as Bond;
try {
  await waitUntilMined(
    await implementationContract.initialize(
      "Placeholder Bond",
      "BOND",
      deployer,
      THREE_YEARS_FROM_NOW_IN_SECONDS,
      "0x0000000000000000000000000000000000000000",
      "0x0000000000000000000000000000000000000001",
      ethers.BigNumber.from(0),
      ethers.BigNumber.from(0),
      0
    )
  );
} catch (e) {
  console.log("Is the contract already initialized?");
  console.log(e);
}
```

Due to the fact that the initially reviewed code did not have the proper initialization for the `Bond` implementation (as it was an unmerged branch) and because in case of a successful exploitation the impact on the system remains minimal, this finding is marked as low risk. It is not necessary to create a separate transaction and initialize the storage of the implementation contract to prevent unauthorized initialization.

Recommendation: OpenZeppelin's minimal proxy pattern implements a more efficient (less gas) and elegant way to lock the implementation contract by simply invoking `_disableInitializers()`, thus this solution is recommended instead of the current mechanism.

Porter: Implemented in PR #262.

Spearbit: Acknowledged, recommendation has been implemented.

5.3 Gas Optimization

5.3.1 Verify amount is greater than 0 to avoid unnecessarily `safeTransfer()` calls

Severity: *Gas Optimization*

Context: Bond.sol#L263

Description: Balance should be checked to avoid unnecessary `safeTransfer()` calls with an amount of `0`.

Recommendation: Implement a check to make sure `amount > 0` to avoid unnecessary `safeTransfer()` calls.

```
        uint256 sweepingTokenBalance = sweepingToken.balanceOf(address(this));
+       if (sweepingTokenBalance == 0) {
+            revert ZeroAmount();
+       }
        sweepingToken.safeTransfer(receiver, sweepingTokenBalance);
```

Porter: Implemented in PR #287.

Spearbit: Acknowledged, recommendation has been implemented.

5.4 Informational

5.4.1 Improve checks for token allow-list

Severity: *Informational*

Context: BondFactory.sol#L93

Description: The `BondFactory` contract has two enabled allow-lists by default, which require the team's approval for issuers and tokens to create bonds. However, the screening process was not properly defined before the assessment.

In case a malicious token and issuer slip through the screening process the protocol can be used by malicious actors to perform mass scam attacks. In such scenario, tokens and issuers would be able to create bonds, sell those anywhere and later on exploit those tokens, leading to loss of user funds.

```
    /// @inheritdoc IBondFactory
    function createBond(
        string memory name,
        string memory symbol,
        uint256 maturity,
        address paymentToken,
        address collateralToken,
        uint256 collateralTokenAmount,
        uint256 convertibleTokenAmount,
        uint256 bonds
    ) external onlyIssuer returns (address clone)
```

Recommendation: Consider

- Ensuring that all tokens are checked and passed through checklist.
- Use automation and scripts to cover this checklist as thoroughly as possible in form of e.g. a report.
- Add logic to support inclusion/exclusion of such tokens or document the non-support warning explicitly to users.

Porter: Since The Team has a checklist for tokens that can be used as a payment or collateral token, this safeguard should act to prevent malicious type of token.

Spearbit: Acknowledged (solution based on procedures, not technical).

5.4.2 Incorrect revert message

Severity: *Informational*

Context: Bond#L81, IBond.sol#14

Description: `error BondBeforeGracePeriodOrPaid()` is used to revert when `!isAfterGracePeriod() && amountPaid() > 0`, which means the bonds is before the grace period and not paid for. Therefore, the error description is incorrect.

```
if (isAfterGracePeriod() || amountUnpaid() == 0) {
    _;
} else {
    revert BondBeforeGracePeriodOrPaid();
}
```

Recommendation: Consider changing this error name to `BondBeforeGracePeriodAndNotPaid()`, and also correct the related Natspec description.

Porter: Implemented in PR #282.

Spearbit: Solution is implemented.

5.4.3 Non-existent bonds naming/symbol restrictions

Severity: *Informational*

Context: BondFactory.sol#L93

Description: The issuer can define any name and symbol during bond creation. Naming is neither enforced nor constructed by the contract and may result in abusive or misleading names which could have a negative impact on the PR of the project.

```
    /// @inheritdoc IBondFactory
    function createBond(
        string memory name,
        string memory symbol,
        uint256 maturity,
        address paymentToken,
        address collateralToken,
        uint256 collateralTokenAmount,
        uint256 convertibleTokenAmount,
        uint256 bonds
    ) external onlyIssuer returns (address clone)
```

A malicious user could hypothetically use arbitrary names to:

- Mislead users into thinking they are buying bonds consisting of different tokens.
- Use abusive names to discredit the team.
- Attempt to exploit the frontend application by injecting arbitrary HTML data.

The team had a discussion regarding naming conventions in the past. However, not all the abovementioned scenarios were brought up during that conversation. Therefore, this finding is reported as informational to revisit and estimate its potential impact, or add it as a test case during the web application implementation.

Recommendation: Consider revisiting the design decision regarding arbitrary names and symbols supplied by end-users in the context of the previously described risks. Especially important is the latest scenario regarding potential HTML injection, which has to be taken into consideration and properly mitigated in web application implementations.

It is recommended to decide on naming conventions and enforce them programmatically, ideally at the smart contract level as any implementation in the UI can be easily bypassed.

Porter: The team understands the risks and potential abuses and will consider it in discussions regarding naming formats. Verified it does not impact on the front-end.

Spearbit: Acknowledged.

5.4.4 Needles variable initialization for default values

Severity: *Informational*

Context: Bond.sol#L333

Description: `uint256` variable are initialized to a default value of `zero` per Solidity docs. Setting a variable to the default value is unnecessary.

Recommendation: Remove explicit initialization for default values.

Porter: Implemented in PR #283.

Spearbit: Acknowledged, solution has been implemented.

5.4.5 Deflationary payment tokens are not handled in the `pay()` function

Severity: *Informational*

Context: Bond.sol#L145

Description: The `pay()` function does not support `rebasing/deflationary/inflationary` payment tokens whose balance changes during transfers or over time. The necessary checks include at least verifying the amount of tokens transferred to contracts before and after the actual transfer to infer any fees/interest.

Recommendation: Consider verifying that the previous balance and the future balance equals to `amount` for any `rebasing/inflation/deflation` reward tokens.

```
+       uint balanceBefore = IERC20Metadata(paymentToken).balanceOf(address(this));

    IERC20Metadata(paymentToken).safeTransferFrom(
            msg.sender,
            address(this),
            amount
        );

+       uint balanceAfter = IERC20Metadata(paymentToken).balanceOf(address(this));
+       uint amountDiff = balanceAfter - balanceBefore;
+       emit Payment(msg.sender, amountDiff);
```

Porter: Implemented in PR #285.

Spearbit: Acknowledged, solution has been implemented.

6 Additional Comments

Additionally, to the audit report a full foundry-based fuzzing testing and CI setup was provided after the assessment to improve the security posture of the project