BarnBridge Smart Yield Bonds Audit

FEBRUARY 26, 2021

The BarnBridge team asked us to review and audit their Smart Yield Bond smart contracts. We looked at the code and now publish our results.

Scope

We audited commit `943df3a8fcd8dd128af3beb0c85a0480c0e95ead` of the BarnBridge/BarnBridge-SmartYieldBonds repository. In scope were the contracts directly inside the `/contracts` folder, however the contracts within `/contracts/mocks` were out of scope. *Note: the repository in question is private at the time of writing, so many hyperlinks will only work for the BarnBridge team.*

Overall Health

In general we found the code to be clearly written, easy to follow, with a fair number of comments. All of these points indicated a healthy and well-written codebase. However, as the audit process progressed, we found that the code, and the continued pace of iteration on the codebase outside of the commit under audit, suggested that the project was not at the finalized stage we normally look for in a healthy audit.

The code we were provided to audit contained an incomplete set of tests, no event emissions, and a number of `TODO` statements that outlined features that had not yet been implemented. During the course of the audit, the team updated features throughout the codebase. The result of these protocol changes is that at the conclusion of our 3 week audit, our review is no longer of the latest version of BarnBridge's Smart Yield protocol.

System Overview

BarnBridge's protocol of Smart Yield Bonds enables risk-averse 'Senior Bond Holders' to protect themselves against the risk of variable rate annuities, such as Compound, going down. A second group of participants referred to as 'Juniors' enable this by providing liquidity and taking on more risk.

Purchasers of Senior Bonds are given a guaranteed return over a fixed period on their principal. Their principal is invested into a protocol with a variable rate of return – currently this is Compound. If a Senior's principal deposit generates a larger return than their guaranteed return, the excess profit is given to the Juniors who bought the risk. Conversely, if the variable rate goes down, then the Juniors will take a loss to satisfy the Seniors' guaranteed return.

An Oracle determines a weighting of Compound's interest in order to determine the guaranteed return a Senior should receive on a given deposit.

Privileged Roles

At deployment, the protocol sets addresses for a `DAO` and `Guardian` for the protocol. The `DAO` has the power to change both the `DAO` and `Guardian` addresses.

A number of actions in the protocol are callable by the `DAO` or the `Guardian`. These include:

- Setting the fee percentage for buying and redeeming tokens
- Setting protocol values such as maximum bond length, and maximum bond rate
- Pausing the buying of Tokens
- Updating the addresses for the oracle, uniswap, bond model, and recipient of protocol fees

Findings

Here we present two client-reported issues, followed by our findings.

Outdated values being used to calculate expected reward

In the `harvest` function, a call to `compRewardExpected` is made to calculate the expected amount of COMP to receive from Compound. Any extra COMP received, above the expected amount, is given to the `feeOwner` as a fee.

However, the expected amount is calculated using an outdated state, and so the result is incorrect. The call to `compRewardExpected` should instead occur *after* the call to `claimComp` so that up to date information is used to calculate the expected amount.

Harvest yield not properly accounted for

The protocol updates time-weighted local state variables to track both accumulated yield in Compound, and extra yield resulting from selling COMP reward tokens. This updating of local variables happens in the `accountYield` modifier, which is applied to a number of functions in the `CompoundProvider` contract.

The protocol mistakenly applies the `accountYield` modifier within the `harvest` function, which was causing these rewards to be incorrectly time-weighted in the state variables.

Critical severity

[C01] Protocol allows non-existent loanable value as Senior Bond gain

When purchasing a new Senior Bond, users call the `buyBond` function, which performs a number of calculations and checks before it either reverts or mints a new Senior Bond NFT to the user. One of the checks performed is that the new Senior Bond's `gain` is less than the protocol's loanable amount.

However, the function that calculates the protocol's loanable amount – `underlyingLoanable` – is incorrect and can underflow. This is due to the fact that the calculation of the Junior Token price via the `price` function uses `abondPaid`, which is the total `abond.gain` accrued so far,

whereas `underlyingLoanable` subtracts the *entire* `abond.gain`. This underflow effectively leads to *any amount* of Senior Bond `gain` being approved, as `underlyingLoanable` outputs values around 1e77.

To illustrate this issue empirically, we have provided a test case in this secret gist. This test can be run after pasting it into line 79 of `jbond.test.ts`.

Consider updating the calculation within `underlyingLoanable`, so that the return value aligns with protocol intentions. Additionally, consider this an excellent example of how utilizing SafeMath could prevent unintentional overflows and underflows that can cause unexpected or undesirable protocol behavior.

Update: *Fixed in PR#42. The referenced PR is based on commits that include various other changes to the codebase which have not been reviewed by OpenZeppelin.*

High severity

[H01] Inconsistent and erroneous handling of ERC20 tokens

The ERC20 token specification defined the required interface of an ERC20 token. This included defining the functions `transfer`, `transferFrom`, and `approve` such that they return a `bool` signalling whether the function was successful or not. The specification goes on to say that functions *should* throw on failure, however throwing is not a requirement. This specification has lead to the development of 3 different types of ERC20 tokens:

1. Tokens that `throw` on failure, and return `true` on success. One example is DAI.
2. Tokens that return `false` on failure, and return `true` on success. One example is ZRX.
3. Tokens that `throw` on failure, and *do not* return a boolean. One example is USDT.

Throughout the codebase, ERC20 token transfers and approvals are handled inconsistently. This leads to a variety of different errors when handling ERC20 tokens. This list is not exhaustive:

- Line 94 of `_takeUnderlying` in the `CompoundProvider` contract reverts if the `transferFrom` returns false. While this works with tokens of types 1 and 2, the `require` statement assumes the transfer is going to return a boolean. This means that for tokens of type 3, the code is trying to read return data that does not exist. This leads to successful transfers of token type 3 reverting. In turn, this means tokens of this type cannot be deposited into the protocol at all.
- Line 213 of `harvest` in the `CompoundProvider` contract does not check the returned result of a `transfer` call. While tokens of type 1 and type 3 would have thrown if the transfer failed, it is vital to check the return value of tokens of type 2. In this example, the protocol assumes it has successfully paid the `msg.sender` of the `harvest` function, when it may have failed and returned false. This issue is also faced when `CompoundProvider`'s _sendUnderlying is called.

Due to the complexities of adapting code for all 3 types of ERC20 tokens, consider instead using the `safeTransfer`, `safeTransferFrom`, and `safeApprove` functions from the OpenZeppelin library `SafeERC20`. This library safely handles ERC20 tokens that behave in different ways, and ensures that all calls revert on failure, whether or not the underlying token does.

Update: *Fixed in PR#41. In the same PR, additional checks were added to verify the transfer amounts of `uTokens`; those checks are not compatible with `uTokens` that assess fees on transfer. The referenced PR is based on commits that include other changes to the codebase which have not been reviewed by OpenZeppelin.*

[H02] Liquidation of a Junior Bond maturity at the time of purchase always reverts

The protocol attempts to ensure that Junior Bonds are always liquidated in the order of their maturation times. The `juniorBondsMaturities` array is used to store the maturity times of Junior Bonds for just this purpose. When a new Junior Bond is added, `_accountJuniorBond` sorts and stores the new bond by maturity into the correct slot in the `juniorBondsMaturities` array.

At the beginning of many function calls, the function `_beforeProviderOp` iterates through the `juniorBondsMaturities` array, starting from the last liquidated maturity, to ensure that all newly matured Junior Bonds are liquidated. The next maturity slot to be liquidated is tracked by the `juniorBondsMaturitiesPrev` variable.

However, in `buyJuniorBond`, if the user mints a Junior Bond that is immediately liquidatable at a previously un-liquidated maturity, then the contract deviates from the above process. This deviation ultimately causes the function call to result in an error, and revert.

The function `buyJuniorBond` executes a call to `_liquidateJuniorsAt` for the maturity slot that corresponds to the newly minted bond, but it does so without first checking that all previous maturities have been liquidated. The `_liquidateJuniorsAt` function proceeds to check that the stored `JuniorBondsAt.price` is 0, and then sets this price to the current protocol price before liquidating the bonds. `_liquidateJuniorsAt` never updates the value of `juniorBondsMaturitiesPrev`.

Next, `buyJuniorBond` proceeds to execute `redeemJuniorBond` with the purpose of burning the Junior Bond that was just purchased. However, when `redeemJuniorBond` executes `_beforeProviderOp`, it begins liquidating all maturities starting at (the never-yet-updated)

`juniorBondsMaturitiesPrev`. When it reaches the maturity for the bond that was liquidated immediately after being minted, the check that the stored `JuniorBondsAt.price` is 0 fails causing the function to revert, because this price was set to a non-0 value earlier when it was liquidated.

Consider updating `buyJuniorBond` to call `_beforeProviderOp` at the beginning of execution. With this in place, `_liquidateJuniorsAt` can then safely increment the value of `juniorBondsMaturitiesPrev` to ensure that all maturity liquidations are attempted only once.

Update: *Fixed in PR#38. The referenced PR is based on commits that include various other changes to the codebase which have not been reviewed by OpenZeppelin.*

[H03] Unchecked complex calculations throughout protocol

Throughout the protocol, calculations are performed without verifying that the values have not overflowed or underflowed. Performing calculations without checking the outputs can lead to severe errors, including the one outlined in the Protocol allows non-existent loanable value as Senior Bond gain issue. While we did not identify other situations in which the protocol state can lead to similar mathematical errors, the calculations present are complex, and we cannot be certain that other examples do not exist.

Given the vital importance of accurate calculations in the protocol, please consider instead implementing calculations using SafeMath, so that underflows and overflows will revert instead of causing cascading errors.

Update: *Fixed in PR#39. The referenced PR is based on commits that include various other changes to the codebase which have not been reviewed by OpenZeppelin.*

Medium severity

[M01] `_beforeProviderOp` susceptible to running out of gas

In `SmartYield`, the `_beforeProviderOp` function runs at the very beginning of most essential, state-altering functions of the protocol. `_beforeProviderOp` also has the potential to be very gas intensive, using at least 13500 gas per iteration when a junior bond maturity needs to be liquidated. As the number of pending liquidations mounts, so do the gas costs associated with taking essentially any action within the protocol. This negative feedback loop could result in the disincentivization of protocol activity, which could, in turn, lead to more time passing and even more junior bonds maturing.

There exists some number of matured junior bonds, with differing maturities, in need of liquidation, that could effectively freeze the protocol. Given the fact that currently on Ethereum's mainnet the block gas limit is 12.5 million, there would need to be roughly 900 junior bond maturities in queue to be liquidated for the `_beforeProviderOp` function to consistently run out of gas. While such a scenario is unlikely to ever manifest itself, if the protocol *were* to enter such a state, it would currently be unable to recover from it.

To nullify this possibility entirely, consider adding an external function that can effectively call `_beforeProviderOp` with a subset of junior bonds.

Update: *Fixed in PR#37. A publicly callable function to liquidate bonds with only a subset of maturities has been added. The referenced PR is based on commits that include other changes to the codebase which have not been reviewed by OpenZeppelin.*

[M02] Users buying Junior Tokens can bypass paying fees

The calculations of fees, such as the fee for buying Junior Tokens, round down due to Solidity truncation. This means that in the case of very small purchases, a fee of 0 will be taken by the protocol. Users can repeatedly call `buyTokens` for small numbers of tokens in each purchase, effectively bypassing the fee paying system.

Considering rounding all calculations in the protocol's favor.

Update: *The BarnBridge team did not address this issue.*

[M03] Lack of event emissions

The codebase is completely devoid of event definitions or emissions. This makes it very difficult for users or other interested parties to track important changes that take place in the system.

Consider emitting events after sensitive changes take place to facilitate tracking and notify off-chain clients that may be following the contracts' activity.

Update: *Partially fixed in PR#21. Several critical system functions do now emit events, but there are still contracts that lack event emissions. For example, no events are emitted by sensitive functions in the `Governed` contract. The referenced PR is based on commits that include other changes to the codebase which have not been reviewed by OpenZeppelin.*

[M04] Senior bond holders not guaranteed their minimum gain

When purchasing a Senior Bond, using the `buyBond` function, a user specifies a `minGain_` for their principal to appreciate by. This allows them to ensure that they are getting an interest rate they are happy with in their bond, before locking their money up for the duration of the bond.

However, after the bond matures and the user calls `redeemBond` to redeem their principal and gain, the user can end up with less `gain` than their originally specified `minGain_`. This is due to the fact that, at this stage, the protocol takes a percentage of the gain as a fee, and the user ends up with a smaller gain than their bond specified.

To avoid misleading users of the protocol, consider ensuring that a user will actually receive their specified `minGain_` from their Senior Bond after all fees are accounted for.

Update: *The BarnBridge team did not address this issue.*

[M05] Rounding errors in `compound` and `compound2`

Due to the fact that Solidity truncates when dividing, performing a division in the middle of a calculation can result in truncated amounts being amplified and these amplifications leading to large errors. In the `MathUtils` contract, the `compound` function divides at the end of every iteration of the `while` loop. The more iterations of the `while` loop, the more the result diverges from the true value. While the `compound2` function is generally more accurate, it also performs this truncated division in the `odd` case of the while loop.

As an example of how significant these errors can be, consider the calculation of compound interest with 100 principal, 1.02e18 rate, and 100 periods. The correct answer for this calculation is `724.46`. `compound` gives the result `584`, and `compound2` gives the result `720`.

Consider updating the calculations to always perform divisions as late as possible, while also ensuring the intermediate results do not overflow.

Update: *The BarnBridge team did not address this issue. They stated: "The code uses only the `compound2` function, which we consider to have a good tradeoff between accuracy/gas*

cost/complexity. Should we decide this is no longer the case, we'll update the `BondModelV1` *which uses the* `compound2` *function."*

[M06] Using stale cToken exchange rate

Throughout the `CompoundProvider` contract, when considering the exchange rate between cTokens and uTokens in a given Compound pool, the pool's `exchangeRateStored` function is used rather than its `exchangeRateCurrent` function. The result is that the exchange rate used is out of date, and relies on parties interacting with the Compound pool incredibly regularly. This leads to incorrect calculations of fees, incorrect calculation of the underlying balance, and incorrect calculation of the expected COMP reward.

Consider using the `exchangeRateCurrent` function to increase the overall accuracy of the protocol.

Update: *Fixed in commit* `6a2b956d6b9df4639358368c93e518ea458f5d68`. *There are no longer any instances of* `exchangeRateStored` *being used in the revised code. The referenced commit includes other changes to the codebase which have not been reviewed by OpenZeppelin.*

Low severity

[L01] BOND_MAX_RATE_PER_DAY doesn't cap bond rates at 30% as intended

The `BOND_MAX_RATE_PER_DAY` variable places an upper bound on the daily interest rate that an oracle can report for an underlying pool. The inline comment alongside this variable states the initial rate assignment should be equal to "APY ~30% / year". However, the calculation used to set this initial value, i.e., `3 * 49201150733 * 5760`, translates to approximately 36.4% APY. Even

though the second part of that computation, `49201150733 * 5760`, would yield an approximately 10% APY, due to compounding, the relationship between this max rate and the resultant APY is superlinear. Consider reconciling the code and comment to avoid any confusion and to clarify intent.

Update: *Fixed in commit `e6ac08cfbe0ab1eb0f682927a11f7e0a5bf20a6b`. The referenced commit includes other changes to the codebase which have not been reviewed by OpenZeppelin.*

[L02] Constants not explicitly declared

Throughout the codebase there are many occurrences of literal values being used with unexplained meaning. This makes areas of the code more difficult to understand. Some examples include:

- The literal value `1e18` is used throughout the codebase to add accuracy to calculations. However, the lack of comments and explanation around this makes it hard to track whether or not a value is correctly scaled at the end of a calculation.
- The literal value `-1` is often used to represent the maximum integer, but this use is generally something that needs to be inferred.
- The literal values `49201150733`, `3` and `5760` used on line 26 of `IController.sol`. While there are comments above this line that give them some context, the source of the values remains unclear.

Literal values in the codebase without an explanation as to their meaning make the code harder to read, understand, and maintain for developers, auditors and external contributors alike.

Consider defining a `constant` variable for every literal value used and giving it a clear and self-explanatory name. Additionally, for complex values, inline comments explaining how they were calculated or why they were chosen are highly recommended.

Update: *Fixed in PR#43. The referenced PR is based on commits that include other changes to the codebase which have not been reviewed by OpenZeppelin.*

[L03] Duplicated code

Throughout the codebase there are examples of duplicated code. Duplicating code can lead to issues later in the development lifecycle and leaves the project more prone to the introduction of errors. Errors can inadvertently be introduced when functionality changes are not replicated across all instances of code that should be identical. Instead of duplicating code, consider having just one function/modifier/contract containing the duplicated code and using *that* whenever the duplicated functionality is required. Examples of duplicated code include, but are not limited to:

- The `SeniorBond` and `JuniorBond` contracts. Consider instead having one common `Bond` contract.
- Conversions from `uToken` to `cToken`, and vice versa throughout `CompoundProvider`. Consider creating helper functions to perform this conversion.
- The check whether `msg.sender == smartYield` is repeated twice in each of `SeniorBond` and `JuniorBond`. Consider having a common modifier to perform this check.
- The majority of the `currentCumulatives` and `_accountYieldInternal` functions are identical. Consider factoring out the identical code into a common helper function.

Update: *Not fixed. The second and fourth points are outside the scope of fix review, given the extensive changes to the relevant contract. The first and third points of this issue were not addressed. In regards to the first point, the BarnBridge team stated: "We decided to keep the two contracts (SeniorBond, JuniorBond) separate, this should give us flexibility for future developments."*

[L04] Inconsistent deletion of redeemed bonds

When a Senior Bond is redeemed, the NFT is burnt, and then the corresponding `seniorBonds` mapping entry is deleted. This means that the details of the Senior Bond can only be found by looking at past blocks. Contrastingly, when Junior Bonds are redeemed, the NFT is burnt, but the corresponding mapping entry is not deleted.

Consider settling on a consistent approach to data integrity when a bond is redeemed.

Update: *Fixed in commit `488eb0b449a56810bb961566f2c828d99959c55c`. The fix was to comment out the relevant senior bond `delete` rather than remove the code entirely. The referenced commit includes other changes to the codebase which have not been reviewed by OpenZeppelin.*

[L05] Inconsistent maturity check

Throughout the majority of the SmartYield contract, a bond is classed as mature when `currentTime >= maturesAt`. One example of this is at the beginning of the `redeemBond` function. However, in the `unaccountBonds` function, the Senior Bond is only classed as mature if `currentTime > maturesAt`, i.e., if the current time is equal to the maturity, then the bond is not processed.

To clarify intent and to reduce possible confusion throughout the codebase, consider using a consistent measure of whether a bond has met maturity.

Update: *Issue fixed in PR#44. The referenced PR includes prior commits that make other changes to the codebase which have not been reviewed by OpenZeppelin.*

[L06] Inconsistent coding style

There are general inconsistencies and deviations from the Solidity Style Guide throughout the codebase. These lead to misconceptions and confusion when reading the code. Below is a non-exhaustive list of inconsistent coding styles observed.

While most `public` and `external` function names do not contain an underscore, some begin with an underscore. This is misleading, as a leading underscore should be reserved for internal functions that are not accessible from a contract's public API. For example:

- The `_depositProvider` function
- The `_sendUnderlying` function

Some `internal` function names start with an underscore, while others do not. For example:

- `_updateCompState` is internal and has a leading underscore.
- `compound2` is internal and does not have a leading underscore.

Most parameters end with an underscore, while some do not. For example:

- The `forInterval` parameter in the `consult` function

- The `yieldCumulativeStart_` parameter in the `computeAmountOut` function

Some global values are defined in all capitals, however this style should be reserved for constants. This can lead users to believe that certain values cannot be changed, when in reality they can be. For example:

- `HARVEST_REWARD` is a non-constant defined in capitals.
- `feesOwner` is a non-constant defined in camel case.

In function definitions split across multiple lines:

- Some put all visibility modifiers on one line, for example in `observationIndexOf`.
- Others split these across different lines, for example in `getUniswapPath`.

Furthermore, function ordering deviates from the recommendation in the Solidity Style Guide. While the codebase does generally adhere to some form of function ordering, it is not always consistently implemented. For example:

- In the `// internals` section of `SmartYield` contract, there is a public function definition for `unaccountBonds`.

Consider enforcing a standard coding style, such as that provided by the Solidity Style Guide, to improve the project's overall legibility. Also consider using a linter like Solhint to define a style and analyze the codebase for style deviations.

Update: *The BarnBridge team did not address this issue.*

[L07] Lack of input validation

While many of the functions in the codebase that take external input do verify those inputs, there are some instances of constructors and functions that lack sufficient input validation. This is especially true of functions that take input from "trusted" parties, such as the setter functions found in `IController` and `CompoundController`, or the constructor of `YieldOracle`. Even a trusted party may make an error while inputting values, and many clients pass along `0` values by default for empty fields. Consider implementing require statements where appropriate to validate *all* user-controlled input.

Update: *The BarnBridge team did not address this issue. They stated: "At this point we prefer not to increase the code complexity by adding validation for trusted party input."*

[L08] Insufficient inline documentation

Although many functions in the codebase are well documented, there are plenty of other functions that are lacking sufficient inline documentation. For example:

- The initialization loop in the `YieldOracle` contract sets `i` to the length of an empty dynamic array rather than to `0` as one might expect. This is, ostensibly, to support an oracle being deployed with a pre-existing set of `yieldObservations`. Intent here should be clarified.
- The comments above `currentCumulatives` describe one of the function's return values, but fail to mention anything about the other return value.
- The internal `_accountJuniorBond` function is critical to the system, yet it lacks even a single line of inline documentation.

Consider adding additional inline documentation throughout the codebase to further clarify intent and improve overall code readability.

Update: *The BarnBridge team did not address this issue.*

[L09] Low test coverage on some contracts

Writing tests is a critical part of the development process of a new protocol, as it allows developers to find bugs in their code before launch. While many of your contracts have 100% test coverage, a number of them have low test coverage, falling as low as 50% in Governed.sol. Consider adding more tests for the contracts, aiming for at least 95% test coverage across all contracts.

Update: *Improved in commit* `688c85a966ba7033230608e1682356d0e40ecd23`*, but still under 95%. The referenced commit includes other changes to the codebase which have not been reviewed by OpenZeppelin.*

[L10] Junior Token holders can manipulate Junior Bond maturity

When buying a Junior Bond, the maturity time for the bond is set to `1 + abond.maturesAt / 1e18` – the user does not get to choose the duration of their Junior Bond. The value of `abond.maturesAt` is updated whenever a user buys or redeems a Senior Bond. This setup means that, by purchasing a Senior Bond with specific parameters, a user can cause the protocol to set a specific value for `abond.maturesAt`. They could then immediately call `buyJuniorBond` to purchase a Junior Bond with their desired duration.

While we have not found a specific attack vector through this style of manipulation, it is important to consider the consequences of users being able to manipulate state variables in this way. Consider updating the formulae for `abond.maturesAt` or `JuniorBond.maturesAt` so that users cannot manipulate the state variables in ways that could potentially give them some advantage.

Update: *The BarnBridge team did not address this issue. They stated: "The ability of new bonds to affect the maturity date of future junior bonds is by design."*

[L11] Unnecessary subtraction obscures past bond state

The `juniorBondsMaturingAt` mapping is used to track all of the Junior Bonds that will be maturing at a given time. This includes tracking the total number of Junior Tokens that are in bonds at this maturity. When Junior Bonds mature, and are liquidated, the total number of tokens that mature at this time is left unchanged. This allows the history of Junior Bonds to be viewed on-chain.

However, when a new Junior Bond is liquidated immediately after minting, as its maturity has already been liquidated, the function `_unaccountJuniorBond` is called. This function subtracts this bond's tokens from the cumulative total stored in the `juniorBondsMaturingAt` mapping. This subtraction uses unnecessary gas, and obscures the state of Junior Bond history that is otherwise preserved in `juniorBondsMaturingAt`.

Consider removing this subtraction to keep the Junior Bond storage accurate.

Update: *The BarnBridge team did not address this issue. They stated: "We process event logs in our backend for historical data."*

Notes & Additional Information

[N01] Assignment in `require` statement

In the `YieldOracle` contract, there is a `require` statement that makes an assignment. This deviates from the standard usage and intention of `require` statements and can easily lead to confusion. To avoid unnecessary confusion, consider moving the assignment to its own line before the `require` statement and then using the `require` statement solely for condition checking.

Update: *Acknowledged, but not fixed. The BarnBridge team has stated that their `YieldOracle` contract is inspired by Uniswap's `ExampleOracleSimple` contract and that they wish for their code to remain as "close to the source" as possible in this case.*

[N02] Commented code

Throughout the codebase there are lines of code that have been commented out with `//`. This can lead to confusion and is detrimental to overall code readability. We have provided a non-exhaustive list of examples below:

- Line 538 of `SmartYield.sol`
- Lines 601 and 603 of `SmartYield.sol`

Consider removing commented out lines of code that are no longer needed.

Update: *Fixed in commit* `025c654d59fcfd290c4730335bc9dd7cdec38048`. *Note: The referenced commit includes other changes to the codebase which have not been reviewed by OpenZeppelin.*

[N03] Convoluted conditionals

The codebase contains some convoluted conditionals that could benefit from simplification. Two examples of this are:

- `if (!(timeElapsed <= windowSize))` could be simplified to `if (timeElapsed > windowSize)`
- `if (!(timeElapsed >= windowSize - periodSize * 2))` could be simplified to `if (timeElapsed < windowSize - periodSize * 2)`

Consider writing conditionals in their simplest form to reduce potential confusion and increase overall code readability.

Update: *Acknowledged, but not fixed. The BarnBridge team has stated that their* `YieldOracle` *contract is inspired by Uniswap's* `ExampleOracleSimple` *contract and that they wish for their code to remain as "close to the source" as possible in this case.*

[N04] Misleading contract naming

The contracts `IProvider`, `IController`, and `IComptroller` are all named following the convention for naming interfaces, despite all being contracts. `IProvider` and `IController` both contain variable declarations and logic that are integral to the system, meaning the contract

names are misleading and decrease code readabiilty. Conversely, `IComptroller` does not define any variables or logic, and could be defined as an interface with external functions.

To increase code readability for future contributors, auditors, and the community, consider renaming `IProvider` and `IController`, and consider making `IComptroller` an interface as its name implies.

Update: *Partially fixed in commit `557395fd0fcf6bc089e10f312b3478b68e08a5ed`. `IComptroller` and `IProvider` were changed to interfaces as their names imply, however `IController` is still a contract. The referenced commit includes other changes to the codebase which have not been reviewed by OpenZeppelin.*

[N05] Unnecessarily convoluted inheritance

There are instances in the codebase where inheritance becomes unnecessarily convoluted. For example, when the abstract contract `IProvider` inherits `IYieldOraclelizable`, it makes the inheritance graph of anything that inherits the former, such as `CompoundProvider`, much more convoluted than necessary. Since the abstract contract `IProvider` isn't implementing any of the interface functions from `IYieldOraclelizable`, consider just inheriting each parent separately in `CompoundProvider`.

As a general rule, consider keeping inheritance as straightforward as possible to increase overall code readability and maintainability.

Update: *Fixed in commit `557395fd0fcf6bc089e10f312b3478b68e08a5ed`. The referenced commit includes other changes to the codebase which have not been reviewed by OpenZeppelin.*

[N06] Gas inefficiencies

Throughout the codebase, there are several opportunities to improve gas efficiency. Below is a lengthy, albeit non-exhaustive, list of such opportunities.

- In `CompoundProvider`, the `_takeUnderlying` function has two `require` statements. The first `require` is unnecessary however, as the second `require` will revert in all cases the first would have.
- In `CompoundProvider`, the `harvest` function makes repeated calls to `ICTokenErc20(cToken).balanceOf(address(this))`. The value could often be cached.
- In `CompoundProvider`, the `harvest` function makes an external call to `IUniswapV2Router.swapExactTokensForTokens` to swap reward tokens for underlying tokens. The "deadline" provided for the swap is `this.currentTime() + 1800`. However, the addition of `1800` is unnecessary since the swap will necessarily resolve in the current block.
- In `CompoundProvider`, the `transferFees` function converts an amount of `cTokens` into `uTokens`, adds the latter onto `underlyingFees`, and then converts that total amount back into `cTokens`. Converting `underlyingFees` to `cTokens` and then adding the two `cToken` values could reduce complexity.
- In `CompoundProvider`, the `_accountYieldInternal` and `currentCumulatives` functions both cast a current blocktime to a `uint32`, but only after performing an unnecessary `mod 2^32`.
- In `MathUtils`, the `compound` and `compound2` functions will use a lot of gas even when `ratePerPeriod` is `0`. There could be a short-circuit to return `principal` in those cases.

- In `SmartYield`, the `bondGain` function makes an external call to `BondModel.gain`. As part of the call, `SmartYield` provides it's own address as the `pool` argument so that `BondModel` can make calls back to get additional information about the `pool`. It would be more efficient to pass along all the `pool` information that `BondModel` requires as part of the first call to `gain`.
- In `SmartYield`, the function `_beforeProviderOp` has a `for` loop where `this.currentTime()` is called *every iteration*. It could be cached.
- Throughout the codebase, there are several instances of external calls made to `currentTime` functions that only return the current `block.timestamp`. Using `block.timestamp` directly would reduce gas costs.
- Throughout the codebase, there are numerous instances of contracts calling local functions via the `this` keyword, which makes those calls external. Instead, by removing the `this.` prefix, these functions can be called internally,. This changes the EVM opcode invocation from `CALL` to `JUMP` and in doing so decreases gas costs. There are times the number of external calls invoked along a call chain are non-trivial. For instance, a call to `buyBond` makes no less than 29 unnecessary external calls:
    - There are 4 unnecessary external calls in the function itself.
    - The nested call to `_beforeProviderOp` performs at least 7 unnecessary external calls.
    - The nested call to `underlyingLoanable` results in at least an additional 7 unnecessary external calls.
    - The nested call to `_mintBond` calls `_accountBond` which then makes 11 unnecessary external calls.
- Unless `msg.sender` needs to be changed to the calling contract, making external calls via `this` inflates gas costs without benefit and should generally be avoided.

Consider optimizing for gas usage wherever possible to improve overall user experience and generally reduce code complexity.

Update: *Partially fixed in commit 025c654d59fcfd290c4730335bc9dd7cdec38048. The `CompoundProvider` contract has been too extensively modified since our original audit to properly facilitate the review of relevant fixes. The sixth, seventh, eighth, and ninth bullets were fixed. The tenth bullet point was partially fixed, with only one unnecessary external call via the `this` keyword remaining. The referenced commit includes various other changes to the codebase which have not been reviewed by OpenZeppelin.*

[N07] Incomplete interface contracts

There are instances in the codebase of interfaces omitting some of the `public` or `external` functions that their corresponding implementation contracts define. For example:

- `IYieldOracle` contains only two function definitions, but `YieldOracle` contains an additional function `observationIndexOf`.
- `ISmartYield` does not list any of the `public` variable getters from `SmartYield`. The `external` function `setup` is also missing from the interface.

Incomplete interfaces may introduce confusion for users, developers, and auditors alike. To improve overall code legibility and minimize confusion, consider modifying the interface contracts to reflect all of the `public` and `external` functions from their respective implementation contracts.

Update: *The BarnBridge team did not address this issue.*

[N08] Inconsistent use of named return variables

There is an inconsistent use of named return variables across the entire codebase. For instance:

- Some functions return named variables.
- Some return explicit values.
- Some declare a named return variable but override it with an explicit return statement.
- Some have explicit return statements that do not return anything.

Consider adopting a consistent approach to return values by removing all named return variables, explicitly declaring them as local variables, and adding the necessary return statements where appropriate. This would improve both the explicitness and readability of the code, and it may also help reduce regressions during future code refactors.

Update: *The BarnBridge team did not address this issue.*

[N09] Naming issues hinder understanding and clarity of the codebase

To favor explicitness and readability, several parts of the contracts may benefit from better naming. Our suggestions are to rename:

- `fractionOf` to `calculatePercentage`. The name `fractionOf` implies calculating `a` as a fraction of `f`.
- `a` to `amount`.
- `f` to `percentage`.
- `cumulativeSecondlyYieldLast` to `latestCumulativeYieldPerSecond`.

- `cumulativeSecondlyYield` to `cumulativeYieldPerSecond`.
- `yieldCumulative` to `cumulativeYieldPerSecond`.
- `b_` to `seniorBond`.
- `d` to `abondDuration`.
- `ts` to `scaledTimestamp`.
- `tmp` to `temp`.
- `_unaccountBond`, `_accountBond`, `_mintBond`, `unaccountBonds`, `redeemBond`, and `buyBond`, to all state `SeniorBond` instead of just `Bond`.
- `setPaused` to `setPausedState`.
- `cTokenBalance` to `userCTokenBalance`.
- `juniorBondsMaturitiesPrev` to `nextJuniorBondToLiquidate`.

Consider renaming these parts of the contracts to increase overall code clarity.

Update: *The BarnBridge team did not address this issue.*

[N10] Comments not following NatSpec

The docstrings of the contracts are not following the Ethereum Natural Specification Format (NatSpec). Consider following this specification on everything that is part of the contracts' public API.

Update: *The BarnBridge team did not address this issue.*

[N11] Unnecessary public visibility in some functions

Throughout the codebase, there are functions that are defined as `public` but are never used locally. Some examples are:

- In `CompoundController`, the `getUniswapPath` function
- In `IController`, the `setHarvestReward` function
- In both `JuniorBond` and `SeniorBond`, the `mint` and `burn` functions
- In `SmartYield`, the `unaccountBonds` function

To favor readability and reduce gas costs, consider reducing the visibility of `public` functions that are not used internally to `external`.

Update: *The BarnBridge team did not address this issue.*

[N12] TODOs in code

There are "TODO" comments in the code base that should be tracked in the project's issues backlog. For instance, on line 615 of `SmartYield.sol`, on line 171 of `CompoundProvider.sol`, and elsewhere. During development, having well described "TODO" comments will make the process of tracking and solving them easier. Without that information, these comments might age, and important information for the security of the system might be forgotten by the time it is released to production.

Consider removing all instances of TODO comments and instead tracking them in the issues backlog. Alternatively, consider linking each inline TODO to the corresponding issues backlog entry.

Update: *Fixed in commit `025c654d59fcfd290c4730335bc9dd7cdec38048`. The referenced commit includes other changes to the codebase which have not been reviewed by OpenZeppelin.*

[N13] Typos

The codebase contains the following typos:

- `curent` should be `current`.
- `totalUndwerlying` should be `totalUnderlying`.
- `withdrawl` should be `withdrawal`.
- `dirrect` should be `direct`.
- `moar` should be `more`.
- `substract` should be `subtract`.
- `colected` should be `collected`.
- `cummulates` should be `cumulates`.
- `IProviderPool` should be `IProvider`.
- `matureing` should be `maturing`.
- `withdrawls` should be `withdrawals`.
- `begginging` should be `beginning`.
- `should start with rewardCToken and with uToken` should be `should start with rewardCToken and end with uToken`.
- `last index of juniorBondsMaturities that was liquidated` should be `next index of juniorBondsMaturities to be liquidated`.

Additionally the protocol specification (out of scope of the audit) contains many typos. Consider correcting these typos to improve code readability.

Update: *Partially fixed in commit `025c654d59fcfd290c4730335bc9dd7cdec38048`. Many of the above typos still exist in the codebase. The referenced commit includes other changes to the codebase which have not been reviewed by OpenZeppelin.*

[N14] Uninformative revert messages in `require` statements

There are several instances in the codebase where `require` statements have ambiguous or imprecise error messages. Below is a non-exhaustive list of identified instances.

- Line 42 of `BondModelV1.sol`
- Line 366 of `CompoundProvider.sol`
- Line 329 of `SmartYield.sol`
- Line 495 of `SmartYield.sol`
- Line 46 of `YieldOracle.sol`

Error messages are intended to notify users about failing conditions, so they should provide enough information so that appropriate corrections can be made to interact with the system. Uninformative error messages greatly damage the overall user experience, thus lowering the system's quality. Therefore, consider not only fixing the specific instances mentioned above, but also reviewing the entire codebase to make sure every error message is informative and user-friendly.

Update: *The BarnBridge team did not address this issue.*

[N15] Unnecessary empty constructor

On line 35 of `IController.sol` there is an unnecessary empty constructor. In the absence of the empty constructor, the `Governed` constructor will still be invoked because of inheritance.

Consider removing empty constructors to improve overall code readability.

Update: *Fixed in commit `025c654d59fcfd290c4730335bc9dd7cdec38048`. The referenced commit includes other changes to the codebase which have not been reviewed by OpenZeppelin.*

[N16] Unused functions

The `sqrt` and `compound` functions of the `MathUtils` library are defined, but they are not used in the codebase.

Consider removing all unused functions to improve the overall maintainabilty and readability of the code.

Update: *The BarnBridge team did not address this issue.*

[N17] Unnecessary imports

The below list outlines contract imports that are unused and are therefore unnecessary.

- `SmartYield.sol` imports `console.sol`
- `SmartYield.sol` imports `IERC20.sol`
- `SmartYield.sol` imports `Governed.sol`
- `SmartYield.sol` imports `IYieldOraclelizable.sol`

Consider removing unused import statements to simplify the codebase and increase overall readability.

Update: *Partially fixed as of commit* `025c654d59fcfd290c4730335bc9dd7cdec38048`. *The* `IERC20` *import still remains. The referenced commit includes other changes to the codebase which have not been reviewed by OpenZeppelin.*

Conclusions

1 critical and 3 high severity issues were found. Some changes were proposed to follow best practices and reduce the potential attack surface.