TranchedPool and BackerRewards

Security Assessment

December 8, 2021

Prepared For:
Blake West | *Goldfinch*
blake@goldfinch.finance

Prepared By:
Josselin Feist | *Trail of Bits*
josselin@trailofbits.com

Alexander Remie | *Trail of Bits*
alexander.remie@trailofbits.com

Changelog:
November 22, 2021: Initial report delivered
December 8, 2021: Fix Log (Appendix C) added

Executive Summary

From November 15 to November 19, 2021, Goldfinch engaged Trail of Bits to review the security of the `TranchedPool` and `BackerRewards` contracts. Trail of Bits conducted this assessment over two person-weeks, with two engineers working from commit `3de8d935` from the private Goldfinch repository.

Trail of Bits focused on gaining an understanding of the system's new features (the `TranchedPool` slice functionality and the payment of GFI reward tokens to backers). We also sought to identify any issues related to those new features or the access controls, asset transfer process, or contracts' composability.

Our review resulted in seven findings ranging from high to undetermined severity. The high-severity finding concerns the incorrect incrementation of the tranche ID counter. In a pool with multiple slices, this issue leads to the duplication of tranche IDs. As a result, the tranches' accesses are incorrect for any pool with multiple slices. Another issue could enable borrowers to prevent lenders from withdrawing their funds. We also found that a borrower will be unable to repay a loan if the pool that issued it does not hold any junior capital. The remaining issues include the risk that lenders and borrowers could collude with each other to secure GFI rewards; the practice of allowing all users to trigger reward withdrawals also poses risks to the protocol.

The new features introduce additional complexity into the system and increase the likelihood of issues. The system would benefit from additional tests, particularly basic unit tests of the behavior of pools with multiple slices.

We recommend that Goldfinch take the following steps before deploying the contracts:

- Address all issues detailed in this report.
- Identify and document the system invariants.
- Implement unit tests and fuzzing of the system invariants.
- Perform a review of the components omitted from this assessment because of time constraints (see the Coverage section).

Project Dashboard

Application Summary

Name	`TranchedPool` and `BackerRewards`
Version	`3de8d935`
Type	Solidity
Platform	Ethereum

Engagement Summary

Dates	November 15–November 19, 2021
Method	Full knowledge
Consultants Engaged	2
Level of Effort	2 person-weeks

Vulnerability Summary

Total High-Severity Issues	1	■
Total Medium-Severity Issues	2	■■
Total Low-Severity Issues	1	■
Total Informational Issues	2	■■
Total Undetermined-Severity Issues	1	■
Total	7	

Category Breakdown

Access Controls	2	■■
Auditing and Logging	1	■
Data Validation	3	■■■
Undefined Behavior	1	■
Total	7	

Code Maturity Evaluation

Category Name	**Description**
Access Controls	**Moderate.** The system would benefit from documentation on its access controls, including on the abilities of all privileged actors. The `TranchedPool` uses multiple-role access controls and sets different privileges across different files and contracts. This pattern increases the access controls' complexity and should be more thoroughly documented in order to prevent mistakes.
Arithmetic	**Further Investigation Required.** Time constraints prevented us from performing an in-depth review of the system's arithmetic. However, the system would benefit from fuzzing and more thorough testing of its arithmetic; this testing should cover the calculation of interest (with a focus on the interest of a multi-slice pool) and the BackerRewards arithmetic.
Assembly Use	**Not Applicable.**
Code Stability	**Moderate.** The codebase is still evolving, and some features were not ready to be reviewed.
Decentralization	**Not Considered.** The decentralization of the system was outside the scope of this engagement.
Upgradeability	**Moderate.** The `BackerRewards` and `TranchedPool` contract use the `delegatecall` pattern to implement upgrades. However, the `TranchedPool` contract includes functions for migrating funds in the event of an incident, so it is unclear why the contract has an upgradeability feature. Moreover, the lack of tests for and documentation on the upgradeability process increases the risk of errors in the process. The system would also benefit from the addition of `slither-check-upgradeability` to the continuous integration pipeline.
Function Composition	**Satisfactory.** The codebase is well structured, and most of the functions are small and have clear purposes.
Front-Running	**Further Investigation Required**. Time constraints prevented us from thoroughly reviewing the system for front-running issues.
Key Management	**Not Considered.** The key management system was outside the scope of this engagement.

Monitoring	**Moderate.** The changes made to the `TranchedPool` contract did not have a significant effect on the contract's events. However, several `BackerRewards` functions do not emit events (TOB-GTPBR-006), and there is no documentation on the use of its events in behavior monitoring.
Specification	**Moderate.** The specification lacks structure and should be standardized and expanded. Additionally, due to the system's complexity, a list of each component's invariants would make it easier to check the contracts' behavior. Lastly, the quality of the code documentation is inconsistent, and several functions lack documentation.
Testing & Verification	**Weak.** The system would benefit from additional testing. Every new feature should have individual tests covering a wide range of scenarios. Testing of the computation of interest, including the interest paid to a pool with slices, would increase users' confidence in the system. Finally, using a fuzzer would help Goldfinch detect issues and reduce the likelihood of compromise.

Engagement Goals

The engagement was scoped to provide a security assessment of the `TranchedPool` and `BackerRewards` contracts.

Specifically, we sought to answer the following questions:

- Is it possible for a backer to withdraw extra rewards?
- Could a junior lender perform operations meant to be performed only by senior lenders?
- Could lenders be prevented from withdrawing their available funds?
- Could borrowers be prevented from making loan payments?
- Could an attacker trap the system?
- Are the access controls implemented correctly?

Coverage

`BackerRewards`. We reviewed the `BackerRewards` contract for flaws that could allow a backer to withdraw more interest than expected. We checked the access controls to ensure that only privileged users can perform critical operations. We also looked for ways to trap funds in the contract or to disrupt its interactions with the `TranchedPool`.

`TranchedPool`. Our review of the `TranchedPool` contract was focused on the modifications made to support the use of multiple slices. We reviewed the use of slices and looked for flaws that could allow an attacker to manipulate deposits or withdrawals or to withdraw more funds than he or she had deposited. We also reviewed the privileges granted to borrowers and checked for ways for a borrower to perform unauthorized actions.

Because of the time constraints of the engagement, we performed only a limited review of the following areas:

- The interest computation and other arithmetic operations in the `BackerRewards` and `TranchedPool` contracts
- The impact of the slice feature on the computation of the interests
- Opportunities for lenders to game the system and secure unwarranted interest
- The `TranchedPool` and `BackerRewards` contracts' composability with other contracts
- The migration functions of the `TranchedPool` contract

These areas would therefore benefit from further review.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

❑ **Increment the tranche counter after the creation of the new junior tranche in `_initializeNextSlice`.** This will ensure that each new tranche is assigned the correct ID. TOB-GTPBR-001

❑ **Make `setAllowedUIDTypes` an owner-only function, and reconsider whether the access controls on `setFundableAt` match the system's intent.** Changing these functions will help prevent malicious borrowers from locking funds in the system. TOB-GTPBR-002

❑ **Have the `BackerRewards._allocateRewards` function return instead of revert if `totalJuniorDeposits > 0`.** That way, a `TranchedPool` will not be trapped if it lacks junior capital. TOB-GTPBR-003

❑ **Monitor lenders and borrowers to ensure that they are not trying to game the system.** Limit the automation of the reward distribution mechanism, and consider requiring that a privileged user validate every GFI token payout. TOB-GTPBR-004

❑ **Either add access controls to `BackerRewards.withdraw` or update the documentation to inform users of the lack of access controls on the reward distribution mechanism.** This will prevent users from losing their rewards due to incorrect integrations with their contracts. TOB-GTPBR-005

❑ **Add events for all operations to strengthen the monitoring and alert systems of the protocol.** TOB-GTPBR-006

❑ **Measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.** TOB-GTPBR-007

Long term

❑ **Thoroughly test every new feature.** This issue would have been caught by testing the behavior of a `TranchedPool` with more than one slice. TOB-GTPBR-001

❑ **Document the abilities, responsibilities, and limitations of all privileged users.** Ensure that the documentation remains up to date, and create tests to check a wide range of scenarios. TOB-GTPBR-002

❑ **Expand the test suite to cover scenarios in which a pool lacks junior or senior capital.** Ensure that all of the system's features behave as expected in those scenarios, and consider using a fuzzer to ensure that lenders can always withdraw their funds. TOB-GTPBR-003

❑ **Analyze the game theory behind the lender–borrower economics, including the ways in which lenders and borrowers can game the system to earn more rewards or to reduce the amount of their payments, respectively.** Be mindful of the risk of lender–borrower collusion and of the fact that a user could be both the borrower and backer of a loan. TOB-GTPBR-004

❑ **Update the documentation of each public/external function to specify its assumptions, behavior, and handling of assets.** This will reduce the likelihood of external integration issues. TOB-GTPBR-005

❑ **Document the expectations surrounding events, and consider using a blockchain-monitoring system to track any suspicious behavior in the contracts.** A monitoring mechanism for critical events would quickly detect any compromised system components. TOB-GTPBR-006

❑ **Monitor the development and adoption of Solidity compiler optimizations to assess their maturity.** TOB-GTPBR-007

Findings Summary

#	Title	Type	Severity
1	Incorrect tranche counter incrementation will break the tranches accesses	Data Validation	High
2	A borrower/locker can trap a depositor’s funds	Data Validation	Medium
3	A lack of junior capital can cause senior capital to become trapped	Data Validation	Medium
4	Lenders and borrowers can secure GFI tokens through collusion	Access Controls	Undetermined
5	Lack of access controls on reward withdrawals	Access Controls	Low
6	Lack of events for critical BackerRewards operations	Auditing and Logging	Informational
7	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational

1. Incorrect tranche counter incrementation will break the tranches accesses

Severity: High
Difficulty: Low
Type: Data Validation
Finding ID: TOB-GTPBR-001
Target: `protocol/core/TranchedPool.sol`

Description

The tranche counter is incremented incorrectly when new slices are created for a pool with at least one existing slice. As a result, all of the tranches in the pool will have incorrect tranche ID, leading to incorrect data access.

Every tranche has a unique ID that indicates its slice and its type (senior or junior). A senior tranche should have an odd number as its ID.

```
function _isSeniorTrancheId(uint256 trancheId) internal pure returns (bool) {
  return trancheId.mod(NUM_TRANCHES_PER_SLICE) == 1;
}
```

Figure 1.1: `core/TranchedPool.sol#L619-L621`

```
// senior tranche ids are always odd numbered
if (_isSeniorTrancheId(trancheInfo.id)) {
  require(hasRole(SENIOR_ROLE, _msgSender()), "Must have SENIOR_ROLE to deposit into the
senior tranche");
}
```

Figure 1.2: `core/TranchedPool.sol#L164-L167`

```
function getTrancheInfo(uint256 trancheId) internal view returns (TrancheInfo storage) {
  require(trancheId > 0 && trancheId <= poolSlices.length.mul(NUM_TRANCHES_PER_SLICE),
"Unsupported tranche");
  uint256 sliceId =
((trancheId.add(trancheId.mod(NUM_TRANCHES_PER_SLICE))).div(NUM_TRANCHES_PER_SLICE)).sub(1);
  PoolSlice storage slice = poolSlices[sliceId];
  TrancheInfo storage trancheInfo = trancheId.mod(NUM_TRANCHES_PER_SLICE) == 1
    ? slice.seniorTranche
    : slice.juniorTranche;
  return trancheInfo;
}
```

Figure 1.3: `core/TranchedPool.sol#L925-L933`

When a new slice is created, two tranches are created (a senior tranche and a junior one, in that order):

```
require(poolSlices.length < 5, "Cannot exceed 5 slices");
TrancheInfo memory seniorTranche = TrancheInfo({
  id: _trancheIdTracker.current(),
  principalSharePrice: usdcToSharePrice(1, 1),
  interestSharePrice: 0,
  principalDeposited: 0,
  lockedUntil: 0
});
_trancheIdTracker.increment();
TrancheInfo memory juniorTranche = TrancheInfo({
  id: _trancheIdTracker.current(),
  principalSharePrice: usdcToSharePrice(1, 1),
  interestSharePrice: 0,
  principalDeposited: 0,
  lockedUntil: 0
});
```

Figure 1.4: `protocol/core/TranchedPool.sol#L675-L690`

The `_trancheIdTracker` assigns a unique ID to every tranche. However, the ID is incremented only between the creation of a senior tranche and the creation of a junior tranche. As a result, the senior tranche of a second slice will have the same ID as the first slice's junior tranche. This means that when it accesses slice information, the contract may access information on the wrong slice; it also means that a new senior tranche will have an ID number identifying it as a junior tranche (and vice versa).

Exploit Scenario
A `TranchedPool` has one slice with a senior tranche (with a tranche ID of ID1) and a junior tranche (ID2). A new slice is created. The new slice has a senior tranche, with an ID of ID2, and a junior tranche, with an ID of ID3. As a result, all the accesses to the junior tranche of the first slice junior's tranche are broken, and refer now to the senior tranche of the second slice. Moreover, the senior tranche of the new slice is identified as a junior tranche, and the junior tranche, as a senior one.

Recommendations
Short term, increment the tranche counter after the creation of the new junior tranche in `_initializeNextSlice`. This will ensure that each new tranche is assigned the correct ID.

Long term, thoroughly test every new feature. This issue would have been caught by testing the behavior of a `TranchedPool` with more than one slice.

2. A borrower/locker can trap a depositor’s funds

Severity: Medium
Difficulty: High
Type: Data Validation
Finding ID: TOB-GTPBR-002
Target: `protocol/core/TranchedPool.sol`

Description
Borrowers can change the list of allowed UID types, preventing depositors from withdrawing their funds.

At any time, a borrower can call the `setAllowedUIDTypes` function to change the list of allowed UID types:

```
function setAllowedUIDTypes(uint256[] calldata ids) public onlyLocker {
  allowedUIDTypes = ids;
}
```

Figure 2.1: `TranchedPool.sol#L142-L144`

A user must have the correct UID type to withdraw funds from a pool:

```
function _withdraw(
  TrancheInfo storage trancheInfo,
  IPoolTokens.TokenInfo memory tokenInfo,
  uint256 tokenId,
  uint256 amount
) internal returns (uint256 interestWithdrawn, uint256 principalWithdrawn) {
  require(config.getGo().goOnlyIdTypes(msg.sender, allowedUIDTypes), "This address has not
been go-listed");
```

Figure 2.2: `TranchedPool.sol#L589-L595`

By changing the UIDs that allow a depositor to make a withdrawal, a borrower can prevent a depositor from withdrawing funds or interest.

The `setFundableAt` function poses a similar problem of lower severity: the owner and borrower of a pool can set `fundableAt` to a time that will prevent the deposit of assets. It is unclear whether this is expected and known behavior.

Exploit Scenario
Alice and Bob provide liquidity to a `TranchedPool`, and Eve borrows funds from it and pays back the loan. Eve has a disagreement with Alice and Bob and calls `setAllowedUIDTypes` to set `allowedUIDTypes` to an empty array. Eve then extorts Alice and Bob, refusing to unlock their funds unless they send her the interest on the loan.

Recommendations

Short term, make `setAllowedUIDTypes` an owner-only function, and reconsider whether the access controls on `setFundableAt` match the system's intent. Changing these functions will help prevent malicious borrowers from locking funds in the system.

Long term, document the abilities, responsibilities, and limitations of all privileged users. Ensure that the documentation remains up to date, and create tests to check a wide range of scenarios.

3. A lack of junior capital can cause senior capital to become trapped

Severity: Medium
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-GTPBR-003
Target: `protocol/core/TranchedPool.sol`

Description

When the loan is repaid, the `TranchedPool` contract calls the `BackerRewards.allocateRewards` function. The function reverts if the `TranchedPool` contract has no junior liquidity. As a result, a pool with no junior liquidity will prevent the loan repayment

The `TranchedPool._assess` function is called to repay a loan, and calls `BackerRewards.allocateRewards:`

```
function _assess() internal {
  [..]

  config.getBackerRewards().allocateRewards(interestPayment);
```

Figure 3.1: `TranchedPool.sol#L1062-L1096`

The `allocateRewards` function calls the internal `_allocateRewards` function, which calls back the `TranchedPool` contract and checks that the pool's junior tranche balance is greater than zero:

```
function _allocateRewards(uint256 _interestPaymentAmount) internal {
  [..]

  uint256 totalJuniorDeposits = pool.totalJuniorDeposits();
  require(totalJuniorDeposits > 0, "Principal balance cannot be zero");
```

Figure 3.2: `BackerRewards.sol#L184-L199`

As a result, if a `TranchedPool` is locked when it does not have any junior capital, the loan cannot be repaid.

Exploit Scenarios

Scenario 1
Eve is the only backer of a junior tranche. She front-runs the calls that will lock the pool and withdraws her funds. The pool is then locked without any junior capital. As a result, the pool's loan repayment is blocked.

Scenario 2
Bob is the borrower of a `TranchedPool`. Bob's pool does not receive a lot of interest, and only funds from the senior pools are received. Bob locks the pool without any junior capital. As a result, Bob is not able to repay the loan.

Recommendations
Short term, have the `BackerRewards._allocateRewards` function return instead of revert if `totalJuniorDeposits > 0`. That way, a `TranchedPool` will not be trapped if it lacks junior capital.

Long term, expand the test suite to cover scenarios in which a pool lacks junior or senior capital. Ensure that all of the system's features behave as expected in those scenarios, and consider using a fuzzer to ensure that lenders can always withdraw their funds.

4. Lenders and borrowers can secure GFI tokens through collusion

Severity: Undetermined
Difficulty: Undetermined
Type: Access Controls
Finding ID: TOB-GTPBR-004
Target: `protocol/core/TranchedPool.sol`

Description
Providers of junior tranche liquidity receive rewards in the form of GFI tokens. These rewards serve as incentives and are paid out from a Goldfinch reserve. However, it is possible to game the system to secure free GFI tokens; to do this, a backer could collude with a borrower, or a borrower could fund his or her own loan.

> Rewards will be granted in GFI from a predefined reserve. Upon each JuniorTranche interest repayment by a borrower to the protocol, we'll distribute rewards to pools, and backers will get a proportionate share.

Figure 4.1: Liquidity Mining/`BackerRewards` Design Documentation

Exploit Scenario
Eve requests a loan for which junior tranche backers will receive GFI tokens worth $1,000. Eve funds the loan and is the sole lender. She then pays back the loan and receives the GFI tokens.

Recommendations
Short term, monitor lenders and borrowers to ensure that they are not trying to game the system. Limit the automation of the reward distribution mechanism, and consider requiring that a privileged user validate every GFI token payout.

Long term, analyze the game theory behind the lender–borrower economics, including the ways in which lenders and borrowers can game the system to earn more rewards or to reduce the amount of their payments, respectively. Be mindful of the risk of lender–borrower collusion and of the fact that a user could be both the borrower and backer of a loan.

5. Lack of access controls on reward withdrawals

Severity: Low
Type: Access Controls
Target: `rewards/BackerRewards.sol`
Difficulty: High
Finding ID: TOB-GTPBR-005

Description
The `BackerRewards` contract allows any user to trigger the withdrawal of the rewards owed to any account. This can break integrations with third-party systems that update their internal balances only after explicit interactions.

```
* @notice PoolToken request to withdraw all allocated rewards
* @param tokenId Pool token id
*/
function withdraw(uint256 tokenId) public {
  uint256 totalClaimableRewards = poolTokenClaimableRewards(tokenId);
  uint256 poolTokenRewardsClaimed = tokens[tokenId].rewardsClaimed;
  IPoolTokens poolTokens = config.getPoolTokens();
  IPoolTokens.TokenInfo memory tokenInfo = poolTokens.getTokenInfo(tokenId);

  address poolAddr = tokenInfo.pool;
  require(config.getPoolTokens().validPool(poolAddr), "Invalid pool!");

  BaseUpgradeablePausable pool = BaseUpgradeablePausable(poolAddr);
  require(!pool.paused(), "Pool withdraw paused");

  ITranchedPool tranchedPool = ITranchedPool(poolAddr);
  require(!tranchedPool.creditLine().isLate(), "Pool is late on payments");

  tokens[tokenId].rewardsClaimed = poolTokenRewardsClaimed.add(totalClaimableRewards);
  safeERC20Transfer(config.getGFI(), poolTokens.ownerOf(tokenId), totalClaimableRewards);
  emit BackerRewardsClaimed(msg.sender, tokenId, totalClaimableRewards);
}
```

Figure 5.1: `rewards/BackerRewards.sol#L160-L181`

Exploit Scenario
Bob uses a smart wallet to handle his pool tokens. The wallet internally tracks the amount of GFI rewards that it receives and updates that amount only when it calls `BackerRewards.withdraw`. Eve calls `BackerRewards.withdraw` on Bob's account, triggering the transfer of his rewards to his wallet. Because Bob's wallet is not aware of the transfer, it does not update the reward balance; as a result, the rewards are lost.

Recommendations
Short term, either add access controls to `BackerRewards.withdraw` or update the documentation to inform users of the lack of access controls on the reward distribution mechanism. This will prevent users from losing their rewards due to incorrect integrations with their contracts.

Long term, update the documentation of each public/external function to specify its assumptions, behavior, and handling of assets. This will reduce the likelihood of external integration issues.

6. Lack of events for critical BackerRewards operations

Severity: Informational
Difficulty: High
Type: Auditing and Logging
Finding ID: TOB-GTPBR-006
Target: `rewards/BackerRewards.sol`

Description
Several critical operations do not emit events. As a result, it will be difficult to review the correct behavior of the contracts once they have been deployed.

The following operations would benefit from triggering events:

- `BackerRewards.setTotalRewards`
- `BackerRewards.setTotalInterestReceived`
- `BackerRewards.setMaxInterestDollarsEligible`

Events generated during contract execution aid in contract monitoring, behavior baselining, and the detection of suspicious activity.

Recommendations
Short term, add events for all operations to strengthen the monitoring and alert systems of the protocol.

Long term, document the expectations surrounding events, and consider using a blockchain-monitoring system to track any suspicious behavior in the contracts. A monitoring mechanism for critical events would quickly detect any compromised system components.

7. Solidity compiler optimizations can be problematic

Severity: Informational
Difficulty: Low
Type: Undefined Behavior
Finding ID: TOB-GTPBR-007
Target: `hardhat.config.base.ts`

Description
The smart contracts have enabled optional Solidity compiler optimizations.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to solc-js—causes a security vulnerability in the contracts.

Recommendations
Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing a system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Testing	Related to test methodology or test coverage
Timing	Related to race conditions, locking, or the order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is relatively small or is not a risk the customer has indicated is important.

Medium	Individual users' information is at risk; exploitation could pose reputational, legal, or moderate financial risks to the client.
High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is commonly exploited; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of a complex system.
High	An attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components
Arithmetic	Related to the proper use of mathematical operations and semantics
Assembly Use	Related to the use of inline assembly
Code Stability	Related to the recent frequency of code updates
Decentralization	Related to the existence of a single point of failure
Upgradeability	Related to contract upgradeability
Function Composition	Related to separation of the logic into functions with clear purposes
Front-Running	Related to resilience against front-running
Key Management	Related to the existence of proper procedures for key generation, distribution, and access
Monitoring	Related to the use of events and monitoring procedures
Specification	Related to the expected codebase documentation
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.)

Rating Criteria	
Rating	**Description**
Strong	The control was robust, documented, automated, and comprehensive.
Satisfactory	With a few minor exceptions, the control was applied consistently.
Moderate	The control was applied inconsistently in certain areas.

Weak	The control was applied inconsistently or not at all.
Missing	The control was missing.
Not Applicable	The control is not applicable.
Not Considered	The control was not reviewed.
Further Investigation Required	The control requires further investigation.

C. Fix Log

On December 3, 2021, Trail of Bits reviewed the fixes and mitigations implemented by the Goldfinch team for findings TOB-GTPBR-003, TOB-GTPBR-005, and TOB-GTPBR-006.

ID	Title	Severity	Status
03	A lack of junior capital can cause senior capital to become trapped	Medium	Fixed (75)
05	Lack of access controls on reward withdrawals	Low	Fixed (79)
06	Lack of events for critical BackerRewards operations	Informational	Fixed (77)