Goldfinch Finance

Security Assessment

February 28, 2022

Prepared for:
Greg Egan
Goldfinch Finance

Prepared by: **Alexander Remie and Spencer Michaels**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and mutually agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As such, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Table of Contents

Executive Summary

Engagement Overview

Goldfinch engaged Trail of Bits to review the security of its Goldfinch Protocol. From February 14 to February 28, 2022, a team of 2 consultants conducted a security review of the client-provided source code, with 4 person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

On March 7, 2022, consultants conducted a complimentary 4-hour review of Goldfinch's fixes for the issues discovered in this report, and found that all of them had been remediated. Details of the fix review are provided in appendix E.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and documentation. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered two significant flaws that impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	1
Informational	1

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	1
Data Exposure	1
Data Validation	2

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-GLDF-3**
  Missing validation of the `vestingInterval` could lead to users losing access to funds that have been granted to them using a vesting schedule. If the `vestingInterval` exceeds the `vestingLength`, all transactions that try to calculate the vesting rewards will revert.

- **TOB-GLDF-4**
  The use of an arbitrary `from` argument for a `transferFrom` of FIDU/USDC allows an attacker to backrun any FIDU/USDC token approvals to the `StakingRewards` contract to steal the approved tokens and create a CurveLP staking position .

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineers were associated with this project:

Alexander Remie, Consultant
alexander.remie@trailofbits.com

Spencer Michaels, Consultant
spencer.michaels@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
February 11, 2022	Pre-project kickoff call
February 24, 2022	Status update meeting #1
February 28, 2022	Report readout meeting; delivery of report draft
March 4, 2022	Fixes provided by client
March 7, 2022	Fix review performed
March 7, 2022	Delivery of fix review

Project Goals

The engagement was scoped to provide a security assessment of the Goldfinch Protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Do any of the infinite-mint prevention changes introduce new bugs?
- Are the infinite-mint prevention changes sufficient to prevent an infinite-mint attack?
- Can a malicious user earn more rewards than he is actually owed?
- Does the zapping from the senior pool to the tranched pools correctly skip fees and slashing of rewards?
- Does the staking of Curve LP tokens work as expected?
- Can a malicious actor steal user funds?
- Can funds get stuck?
- Are the pro rated backer rewards calculated correctly?
- Are the backer rewards accumulators used correctly?

Project Targets

The engagement involved a review and testing of the following repository.

feb-14-audit branch of the Goldfinch monorepo

Repository	https://github.com/warbler-labs/mono
Version	`d47cdc2d34e927fadec2ac8c5ba0283b72197206`
Type	Smart Contract
Platform	Solidity

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. We focused our audit on code related to the following pull requests, each of which introduced new features or protocol changes.

- Curve USDC-FIDU staking, which augments the `StakingRewards` contract to accept FIDU-USDC Curve LP tokens rather than just FIDU.

- Zapper, which allows no-cost capital moves from the senior pool to tranched pools.

- Backer staking rewards, which allows backers to earn additional rewards equivalent to what they would have received by staking in the senior pool.

- Several small PRs that:

    - Prevent senior-tranche pool tokens from participating in backer rewards

    - Eliminate an edge case that would allocate an excessive amount of rewards

    - Avoid reverting when fractional tokens are processed

    - Allow partial vesting periods

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- One pull request, #338, was merged on the second-to-last day of the project, with the understanding that engineers could review it only briefly, on a best-effort basis. Owing to time constraints, engineers were not able to review this PR substantially.

Automated Testing Results

Trail of Bits has developed three unique tools for testing smart contracts. Descriptions of these tools and details on the use of tools in this project are provided below.

- Slither is a static analysis framework that can statically verify algebraic relationships between Solidity variables. We used Slither to detect possible vulnerabilities.

- Echidna is a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation. We used Echidna to validate the correctness of the vesting reward arithmetic.

Automated testing techniques augment our manual security review but do not replace it. Each technique has limitations: Slither may identify security properties that fail to hold when Solidity is compiled to EVM bytecode, and Echidna may not randomly generate an edge case that violates a property.

We follow a consistent process to maximize the efficacy of testing security properties. When using Echidna, we generate 10,000 test cases per property; and we then manually review the results.

Our automated testing and verification focused on the following system properties:

CommunityRewardsVesting. Library used to calculate the vesting of grants.

Property	Tool	Result
`getTotalVestedAt` does not revert for existing grants.	Echidna	TOB-GLDF-3

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Consistent use of SafeMath prevents underflows/overflows. However, we uncovered one bug due to an unexpected underflow which could revert certain vesting reward related transactions.	Moderate
Auditing	Events are not emitted for some exceptional cases.	Moderate
Authentication / Access Controls	The system correctly applies access controls to all functions.	Satisfactory
Complexity Management	The system is divided into several contracts that each deal with a particular facet of the system. Functions are small and perform a single specific task.	Satisfactory
Cryptography / Key Management	API keys are hardcoded or stored in environment variables.	Weak
Decentralization	This was not part of the scope of this assessment.	Not Considered
Documentation	Each major pull request provided by the client was accompanied by a detailed design document.	Strong
Front-Running Resistance	A backrunning issue was discovered that allows an attacker to steal funds that have been approved for staking.	Weak
Low-Level Calls	No low-level calls were used in the audited code.	Satisfactory
Testing and Verification	The project has unit tests, but coverage is incomplete.	Moderate

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Exposed secrets	Data Exposure	Medium
2	Missing events for early-return scenarios	Auditing and Logging	Informational
3	Missing validation of vestingInterval could lead to vesting DoS	Data Validation	High
4	Arbitrary staker in depositToCurveAndStakeFrom allows attacker to steal any approved FIDU/USDC tokens in StakingRewards	Data Validation	High

Detailed Findings

1. Exposed Secrets	
Severity: **Medium**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-GLDF-1
Target: `packages/protocol/blockchain_scripts/deployHelpers/index.ts:23` `packages/client/src/ethereum/networkMonitor.ts:12`	

Description

Two API keys are hardcoded into the Goldfinch project, one for the Defender admin client and one for the network monitor notification system. In the event of a source code leak or a local-file-read vulnerability, an attacker could obtain these keys and thereby gain privileged access to the services in question.

```
const DEFENDER_API_KEY = process.env.DEFENDER_API_KEY || "A2Ug<...>9C6d"
```

Figure 1.1: `index.ts:23`

```
const NOTIFY_API_KEY = "8447<...>1982"
```

Figure 1.2: `networkMonitor.ts:12`

Recommendations

Short term, rotate the exposed keys immediately. Remove the default key option from `index.ts:23` and replace the fixed key in `networkMonitor.ts:12` with `process.env.NOTIFY_API_KEY`.

Long term, ensure that secrets are never hardcoded, and consider retrieving sensitive values from a secret store such as HashiCorp Vault rather than from environment variables (which can be read by other processes on the same system).

2. Missing events for early-return scenarios

Severity: **Informational**	Difficulty: **High**
Type: Auditing and Logging	Finding ID: TOB-GLDF-2
Target: `packages/protocol/contracts/rewards/BackerRewards.sol` `packages/protocol/contracts/rewards/StakingRewards.sol`	

Description

The two early-return if clauses do not emit an event. Both of these cases are unlikely to happen. However, if they do happen the Goldfinch team will not be notified and can not; if deemed necessary; take appropriate action.

```
if (totalJuniorDepositsAtomic < _gfiMantissa()) {
  return;
}
```

Figure 2.1: BackerRewards.sol:400-402

```
if (additionalRewardsPerToken > rewardsSinceLastUpdate) {
  return 0;
}
```

Figure 2.2: StakingRewards.sol:217-219

```
if (
  newAccumulatedRewardsPerTokenAtLastWithdraw <
  tokenStakingRewards[tokenId].accumulatedRewardsPerTokenAtLastWithdraw
) {
  return;
}
```

Figure 2.3: BackerRewards.sol:530-535

```
if (newStakingRewardsAccumulator <
poolInfo.accumulatedRewardsPerTokenAtLastCheckpoint) {
  return;
}
```

Figure 2.4: BackerRewards.sol:441-443

Each of the above code snippets are used to catch the case where the denominator is less than 1. In that case the calculation will be flawed and return a higher result then should be possible, a.k.a. “infinite mint”. As such, if any of the above if bodies get executed it indicates there could be a problem within the specific rewards contract, and further investigation by Goldfinch is recommended.

Recommendations

Short term, emit an event in each of the four if-bodies. This event can then be monitored using blockchain monitoring services.

Long term, consider using a blockchain-monitoring system to track important system events that could indicate problems within the system. By setting up such a system the Goldfinch team will automatically be notified when important events are emitted that require a closer investigation.

3. Missing validation of vestingInterval could lead to vesting DoS	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-GLDF-3
Target: `packages/protocol/contracts/rewards/CommunityRewards.sol`	

Description

Missing validation of the `vestingInterval` will block the vesting reward calculation if `vestingInterval` is greater than `vestingLength`. The blocking happens because of an underflow which will revert the transaction. For any account where this is the case the vesting rewards are indefinitely locked as the reward calculation function always reverts and therefore no rewards can be claimed.

```
function grant(
  address recipient,
  uint256 amount,
  uint256 vestingLength,
  uint256 cliffLength,
  uint256 vestingInterval
) external override nonReentrant whenNotPaused onlyDistributor returns (uint256
tokenId) {
  return _grant(recipient, amount, vestingLength, cliffLength, vestingInterval);
}

function _grant(
  address recipient,
  uint256 amount,
  uint256 vestingLength,
  uint256 cliffLength,
  uint256 vestingInterval
) internal returns (uint256 tokenId) {
  require(amount > 0, "Cannot grant 0 amount");
  require(cliffLength <= vestingLength, "Cliff length cannot exceed vesting
length");
  require(amount <= rewardsAvailable, "Cannot grant amount due to insufficient
funds");

  if (vestingInterval == 0) {
    vestingInterval = vestingLength;
  }

  rewardsAvailable = rewardsAvailable.sub(amount);
```

```
  _tokenIdTracker.increment();
  tokenId = _tokenIdTracker.current();

  grants[tokenId] = CommunityRewardsVesting.Rewards({
    totalGranted: amount,
    totalClaimed: 0,
    startTime: tokenLaunchTimeInSeconds,
    endTime: tokenLaunchTimeInSeconds.add(vestingLength),
    cliffLength: cliffLength,
    vestingInterval: vestingInterval,
    revokedAt: 0
  });

  _mint(recipient, tokenId);

  emit Granted(recipient, tokenId, amount, vestingLength, cliffLength,
vestingInterval);

  return tokenId;
}
```

Figure 3.1: CommunityRewards.sol:154-199

Figure 3.1 shows the `_grant` function which contains no validation of `vestingInterval`. It does contain a special case (highlighted) of setting `vestingInterval` to `vestingLength` if the passed in `vestingInterval` value is zero.

```
function getTotalVestedAt(
  uint256 start,
  uint256 end,
  uint256 granted,
  uint256 cliffLength,
  uint256 vestingInterval,
  uint256 revokedAt,
  uint256 time
) internal pure returns (uint256) {
  if (time < start.add(cliffLength)) {
    return 0;
  }

  if (end <= start) {
    return granted;
  }

  uint256 elapsedVestingTimestamp = revokedAt > 0 ? Math.min(revokedAt, time) :
time;
  uint256 elapsedVestingUnits =
(elapsedVestingTimestamp.sub(start)).div(vestingInterval);
  uint256 totalVestingUnits = (end.sub(start)).div(vestingInterval);
  return Math.min(granted.mul(elapsedVestingUnits).div(totalVestingUnits), granted);
}
```

Figure 3.2: CommunityRewardsVesting.sol:58-79

Figure 3.2 shows the `getTotalVestedAt` function that is used to calculate the vesting rewards of a given grant. The highlighted line shows the line where the underflow will happen if `vestingInterval > vestingLength`.

```
test_vest(uint32,uint32):  failed!💥
  Call sequence:
    test_vest(0,100000001)
```

Figure 3.3: Echidna test failure

Figure 3.3 shows an Echidna test failure due to an unexpected revert in `getTotalVestedAt`. The triggered revert is an underflow in the SafeMath library, specifically the `div` function. The full Echidna test file can be found in Appendix D.

A grant can only be created by an account that has the "distributor" role. This will be the `MerkleDistributor` contract. This contract lets users submit merkle proofs for grants, that after validation will be created in the `CommunityRewards` contract. It is therefore unlikely that users will receive a signature from Goldfinch where the signed data includes a `vestingInterval` that is greater than `vestingLength`.

When a grant is created in the `CommunityRewards` contract the total rewards available is reduced by the total grant amount. The admin of the contract can at any time deposit more funds into the contract, which will increase the total rewards available. Taking these two together, if a grant is invalid due to a too big `vestingInterval`, then those funds will be locked in the contract as they were already subtracted from the total rewards available.

Lastly, in case the described issue happens Goldfinch could develop an upgrade to the contract that retrieves access to the stuck funds. However, when possible, upgrading the contract should be prevented as it is complex and error-prone.

Exploit Scenario

Alice, a member of the Goldfinch team, deploys a `CommunityRewards` contract with a merkle tree that encodes a vesting where the `vestingInterval` is greater than `vestingLength`. Bob calls `acceptGrant` passing in his `merkleProof` and the grant is created. Bob calls `getReward` after some of the grant has vested, but the function reverts. Bob cannot claim any of his vested grant rewards.

Recommendations

Short term, add a check that reverts the creation of a grant if `vestingInterval` is greater than `vestingLength`.

Long term, make use of property based testing with Echidna to check that important functions do not cause unexpected reverts. This helps to catch unforeseen cases where a revert causes a DoS in (parts of) the contract(s).

4. Arbitrary staker in depositToCurveAndStakeFrom allows attacker to steal any approved FIDU/USDC tokens in StakingRewards	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-GLDF-4
Target: `packages/protocol/contracts/rewards/StakingRewards.sol`	

Description

Allowing an arbitrary `staker` in the `depositToCurveAndStakeFrom` lets an attacker create CurveLP staking positions from any approved FIDU/USDC tokens. These CurveLP tokens can then immediately be withdrawn by an attacker.

```
function depositToCurveAndStake(uint256 fiduAmount, uint256 usdcAmount) public {
  depositToCurveAndStakeFrom(msg.sender, msg.sender, fiduAmount, usdcAmount);
}

/// @notice Deposit to FIDU and USDC into the Curve LP, and stake your Curve LP
tokens in the same transaction.
/// @param fiduAmount The amount of FIDU to deposit
/// @param usdcAmount The amount of USDC to deposit
function depositToCurveAndStakeFrom(
  address staker,
  address nftRecipient,
  uint256 fiduAmount,
  uint256 usdcAmount
) public nonReentrant whenNotPaused updateReward(0) {
  require(fiduAmount > 0 || usdcAmount > 0, "Cannot stake 0");

  IERC20withDec usdc = config.getUSDC();
  IERC20withDec fidu = config.getFidu();
  ICurveLP curveLP = config.getFiduUSDCCurveLP();

  // Transfer FIDU and USDC from staker to StakingRewards, and allow the Curve LP
contract to spend
  // this contract's FIDU and USDC
  if (fiduAmount > 0) {
    fidu.safeTransferFrom(staker, address(this), fiduAmount);
    fidu.safeIncreaseAllowance(address(curveLP), fiduAmount);
  }
  if (usdcAmount > 0) {
    usdc.safeTransferFrom(staker, address(this), usdcAmount);
    usdc.safeIncreaseAllowance(address(curveLP), usdcAmount);
```

```
  }

  // Calculate the expected number of Curve LP tokens to receive
  uint256 expectedAmount = curveLP.calcTokenAmount([fiduAmount, usdcAmount], true);

  // Add liquidity to Curve. The Curve LP tokens will be minted under StakingRewards
  uint256 curveLPTokens = curveLP.addLiquidity([fiduAmount, usdcAmount],
expectedAmount, address(this));

  // Stake the Curve LP tokens on behalf of the user
  uint256 tokenId = _stakeWithLockup(
    address(this),
    nftRecipient,
    curveLPTokens,
    StakedPositionType.CurveLP,
    0,
    MULTIPLIER_DECIMALS
  );

  emit DepositedToCurveAndStaked(msg.sender, fiduAmount, usdcAmount, tokenId,
curveLPTokens, 0, MULTIPLIER_DECIMALS);
}
```

Figure 4.1: StakingRewards.sol:424-471

Figure 4.1 shows the `depositToCurveAndStake` and `depositToCurveAndStakeFrom` functions. The highlighted lines show where the arbitrary staker argument is used to transfer FIDU/USDC tokens into the `StakingRewards` contract.

All of the deposit functions in the StakingRewards contracts require that a user has previously approved FIDU/USDC to the StakingRewards contract. By allowing an arbitrary `staker` argument in `depositToCurveAndStakeFrom` an attacker could monitor for any token approval calls and call `depositToCurveAndStakeFrom` with `staker` set to the approval's initiator and `nftRecipient` to the attacker's address. This will then use the approved tokens of the victim to deposit into Curve and use the Curve LP tokens to create a position for the attacker (`nftRecipient`). The attacker can then immediately unstake this position to retrieve the CurveLP (USDC-FIDU) tokens.

```
/// @notice Zap staked FIDU into staked Curve LP tokens without losing unvested
rewards
///  or paying a withdrawal fee.
/// @param tokenId A staking position token ID
/// @param fiduAmount The amount in FIDU from the staked position to zap
function zapStakeToCurve(uint256 tokenId, uint256 fiduAmount) public whenNotPaused
nonReentrant {
  IStakingRewards stakingRewards = config.getStakingRewards();
  require(IERC721(address(stakingRewards)).ownerOf(tokenId) == msg.sender, "Not
token owner");

  uint256 stakedBalance = stakingRewards.stakedBalanceOf(tokenId);
```

```
  require(fiduAmount <= stakedBalance, "cannot unstake more than staked balance");

  stakingRewards.unstake(tokenId, fiduAmount);

  SafeERC20.safeApprove(config.getFidu(), address(stakingRewards), fiduAmount);

  stakingRewards.depositToCurveAndStakeFrom(address(this), msg.sender, fiduAmount,
0);
}
```

Figure 4.2: `Zapper.sol:121-137`

The Zapper contract uses the `depositToCurveAndStakeFrom` to create a staking position for an account that wants to convert (part of) their staked FIDU into a deposit of Curve LP tokens (see Figure 4.2). To make this work it sets the `staker` to the `Zapper` contract, and the `nftRecipient` to the caller. This makes part of the arbitrary arguments of `depositToCurveAndStakeFrom` a necessity, namely the `nftRecipient`. However, if the `depositToCurveAndStakeFrom` was adjusted to not allow an arbitrary `staker` and instead used `msg.sender`, than the `Zapper` contract would still be able to function correctly, as the only required arbitrary argument is the `nftRecipient`.

Exploit Scenario

Alice approves 1000 FIDU to the `StakingRewards` contract, to be used for a subsequent call to stake the tokens. Eve spots this transaction in the mempool and backruns it with a call to `depositToCurveAndStakeFrom` with `staker` set to Alice and `nftRecipient` set to Eve, and `fiduAmount` set to 1000. Eve now owns a staking position with the 1000 FIDU that was approved by Alice. Eve unstakes the position and receives CurveLP tokens, which she immediately swaps back to FIDU tokens.

Recommendations

Short term, remove the staker argument from `depositToCurveAndStakeFrom` and instead use `msg.sender` as `staker`.

Long term, prevent the use of an arbitrary `from` address in `transferFrom` calls as these can almost certainly be exploited by an attacker that monitors token approvals.

Summary of Recommendations

The additions to the Goldfinch smart contracts are a work in progress with multiple planned iterations. Trail of Bits recommends that the Goldfinch team address the findings detailed in this report and take the following additional steps prior to deployment:

- Set up a blockchain monitoring system that keeps track of important emitted events that could indicate (a) problem(s) in the contracts.
- Expand the unit test suite coverage, and specifically add tests for the validation of the vesting interval (TOB-SHER-3).
- Create important system invariants and test them using Echidna. This will both ensure that the test suite is robust and uncover edge cases that are not covered by the unit test suite.
- Ensure the entire Goldfinch protocol has been security reviewed. The system is still being actively developed and certain parts have not yet been security reviewed. For example: #338.
- Update the user-facing documentation to include the changes that are part of the reviewed PRs.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

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

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Calls	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of future vulnerabilities.

- Update the outdated comment at line 153 of `packages/protocol/contracts/rewards/CommunityRewards.sol`, which refers to a `require()` that has since been removed.
- Save gas by converting the functions `_gfiMantissa()`, `_fiduMantissa()`, and `_usdcMantissa()` at lines 629-639 in `packages/protocol/contracts/rewards/BackerRewards.sol` to constants.
- Ensure that all internal functions are consistently prepended with an underscore to improve readability. For instance, `depositToSeniorPool()` at line 397 of `packages/protocol/contracts/rewards/StakingRewards.sol` does not follow this pattern.

D. Echidna test vesting rewards

This section shows the Echidna test that was used for finding the unexpected revert in `getTotalVestedAt`. Note that we rewrote the `getTotalVestedAt` function to make it easier to test standalone using Echidna, and that we perform the vesting arithmetic in a separate contract (`VestingCalc`) to allow using a try-catch to test for unexpected reverts. Lastly, we added a `test_vest_fixed` function that contains the correct validation of `vestingInterval`, and that does not unexpectedly revert.

```
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import "@openzeppelin/contracts-ethereum-package/contracts/math/Math.sol";
import "@openzeppelin/contracts-ethereum-package/contracts/math/SafeMath.sol";

contract VestingCalc {
  using SafeMath for uint;

  function calcVested(uint currentTime, uint startTime, uint endTime, uint
vestingInterval, uint amount) external pure returns (uint granted) {
    if (currentTime < startTime) {
      return 0;
    }

    if (endTime <= startTime) {
      return granted;
    }

    uint elapsedVestingUnits = currentTime.sub(startTime).div(vestingInterval);
    uint totalVestingUnits = endTime.sub(startTime).div(vestingInterval);

    granted = Math.min(amount.mul(elapsedVestingUnits).div(totalVestingUnits),
amount);
  }
}

contract VestingEchidna {
    using SafeMath for uint;

    VestingCalc calcContract;

    event AssertionFailed(
        string reason
    );

    constructor() public {
      calcContract = new VestingCalc();
    }
```

```
function test_vest(
    uint32 timePassed,
    uint32 vestingInterval
) public {
    uint startTime =   1_600_000_000;
    uint vestingLength = 100_000_000;

    uint amount = 100e18;

    require(vestingInterval != 0);
    require(amount > 0);

    uint currentTime = startTime.add(uint(timePassed));

    try calcContract.calcVested(
      currentTime,
      startTime,
      startTime.add(vestingLength),
      uint(vestingInterval),
      uint(amount)
    ) returns (uint granted) {
      // success
    } catch Error(string memory reason) {
      emit AssertionFailed(
        reason
      );
    }
}

function test_vest_fixed(
    uint32 timePassed,
    uint32 vestingInterval
) public {
    uint startTime =   1_600_000_000;
    uint vestingLength = 100_000_000;

    uint amount = 100e18;

    require(vestingInterval != 0);
    require(amount > 0);

    require(vestingInterval <= vestingLength); // <-- the FIX

    uint currentTime = startTime.add(uint(timePassed));

    try calcContract.calcVested(
      currentTime,
      startTime,
      startTime.add(vestingLength),
      uint(vestingInterval),
      uint(amount)
    ) returns (uint granted) {
```

```
            // success
          } catch Error(string memory reason) {
            emit AssertionFailed(
              reason
            );
          }
      }
}
```

Figure D.1: Echidna test for `getTotalVestedAt`

E. Fix Log

On March 7, 2022, Trail of Bits reviewed the fixes and mitigations implemented by the Goldfinch team for the issues identified in this report. All four of the the issues reported in the original assessment were fixed, as were the three code quality recommendations in appendix C. We reviewed each of the fixes to ensure that the proposed remediation would be effective. For additional information, please refer to the Detailed Fix Log section below.

ID	Title	Severity	Fix Status
1	Exposed Secrets	Medium	Fixed
2	Missing events for early-return scenarios	Informational	Fixed
3	Missing validation of vestingInterval could lead to vesting DoS	High	Fixed
4	Arbitrary staker in depositToCurveAndStakeFrom allows attacker to steal any approved FIDU/USDC tokens in StakingRewards	High	Fixed

Detailed Fix Log

TOB-GLDF-1: Exposed secrets
Fixed in `a29d909`. The default `DEFENDER_API_KEY` value was removed and rotated. The exposure of `NOTIFY_API_KEY` was confirmed as not of concern, as it is deployed to the client, and thus effectively public.

TOB-GLDF-2: Missing events for early-return scenarios
Fixed in `2ca3c96`. The missing events were added.

TOB-GLDF-3: Missing validation of vestingInterval could lead to vesting DoS
Fixed in `e9ef135`. A `require()` was added to guarantee that `vestingInterval` does not exceed `vestingLength`.

TOB-GLDF-4: Arbitrary staker in depositToCurveAndStakeFrom allows attacker to steal any approved FIDU/USDC tokens in StakingRewards
Fixed in `fa513b2e`. The `staker` parameter was removed from the function signature of `depositToCurveAndStakeFrom()`, and all uses of `staker` within that function were replaced with `msg.sender`.