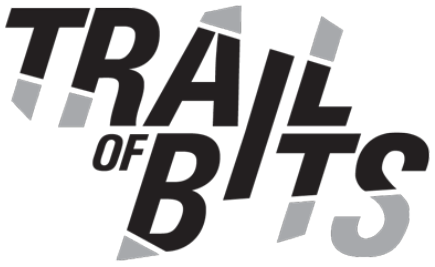

IDEO CoLabs - Goldfinch

Security Assessment

September 7, 2021

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

Prepared By:
Devashish Tomar | *Trail of Bits*
devashish.tomar@trailofbits.com

Natalie Chin | *Trail of Bits*
natalie.chin@trailofbits.com

Executive Summary

From August 16, 2021 through September 3, 2021 IDEO CoLabs engaged Trail of Bits to review the security of Goldfinch V2. Trail of Bits conducted this assessment over the course of 6 person-weeks with 2 engineers working from fc70bbf from the goldfinch-contracts repository.

During the first week of the engagement, we focused on gaining an understanding of the Goldfinch system and architecture and deployment scripts. During the second week, we focused on the `StakingRewards` contract and the `TranchedPool` and `CreditLine` logic, with a focus on interest rate calculations, difference between expected and desired share price, and what happens when loans default. During the final week, we focused on the system arithmetic, distribution of funds between the junior and senior tranches, and corner-cases associated with loan repayment.

Our review resulted in 19 findings, ranging from high to informational severity. One high-severity issue was reported, which would allow an attacker a low-cost griefing attack on the `SeniorPool`, to prevent it from funding the junior and senior tranches of TranchedPool. Two additional high-severity issues were raised, regarding initialization functions being front-run and a lack of contract existence check. These issues would result in the Goldfinch system being misconfigured, and code being assumed to be executed. We also raised two medium-severity issues, regarding insufficient protection on sensitive keys and project dependencies containing vulnerabilities. Additional informational-severity issues were raised around overly-complicated architecture and lack of documentation.

Appendix C outlines code quality recommendations that are not related to any particular security issue. Appendix D outlines the assumptions that were made regarding the Goldfinch-deployed environment. Appendix E provides guidance for interacting with arbitrary tokens. Appendix F provides a Slither script meant to help ensure that all functions are protected by the `onlyAdmin` modifier when necessary.

Overall, the Goldfinch V2 codebase is an extremely complex system with many interacting components. Trail of Bits was provided with insufficient documentation on system architecture and mathematical derivations. This lack of specifications made it significantly more difficult to review the code, as code invariants, properties, and expected behavior was very unclear.

Before any further deployment, Trail of Bits recommends Goldfinch to address the concerns highlighted in this report, in addition to focusing on improving documentation on system architecture, flowcharts and the movement, flow of funds, pre- and post-conditions of functions, and identifying system invariants throughout the system. The codebase would also benefit from code simplifications, including:

- The reduction of the inheritance tree.
- The implementation of the arithmetics through a 1-1 match between their specification and the code.
- Re-architecting functions to allow for simplified nested function calls instead of being spread across inherited contracts.

Reducing the unnecessary code complexity, and striving for simplicity will ease the code review, and reduce the likelihoods of vulnerabilities.

Project Dashboard

Application Summary

Name	Goldfinch V2
Version	fc70bbf
Type	Solidity
Platforms	Ethereum

Engagement Summary

Dates	August 16, 2021 - September 3, 2021
Method	Full Knowledge
Consultants Engaged	2
Level of Effort	6 person-weeks

Vulnerability Summary

Total High-Severity Issues	3	■■■
Total Medium-Severity Issues	2	■■
Total Low-Severity Issues	4	■■■■
Total Informational-Severity Issues	9	■■■■■■■■■
Total Undetermined-Severity Issues	1	■
Total	19	

Category Breakdown

Access Controls	1	■
Authentication	1	■
Configuration	8	■■■■■■■■
Data Validation	5	■■■■■
Documentation	1	■
Error Reporting	1	■
Patching	2	■■
Total	19	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Moderate.** There are a lot of privileged actors used throughout this codebase, which are not clearly identified and often lacking documentation. While the roles are split and named differently, the codebase would benefit significantly from the documentation of the purposes of these roles.
Arithmetic	**Weak.** While we did not find an issue related to arithmetic, the logic in the system is extremely complicated. The lack of documentation hindered the audit and general understanding of the arithmetic in the codebase, making it significantly more difficult to determine unexpected inputs.
Assembly Use/Low-Level Calls	**Weak.** The lack of contract existence check in the proxies (TOB-GFH-005) contract can result in code silently failing. Additionally, the codebase contains a significant amount of assembly throughout, which lacks thorough documentation.
Decentralization	**Weak.** The codebase provides the owner of the contract a significant amount of privilege for adding new Borrowers into a system, through a centralized KYC system provided by Goldfinch. While this iteration is not intended to be decentralized, the whitepaper outlines a fully decentralized future iteration. We were not provided with user documentation outlining user risks, deployment risks, or how users would verify a Goldfinch-deployed contract.
Code Stability	**Strong.** The code did not change during the audit.
Contract Upgradeability	**Weak.** All contracts except `FixedLeverageRatioStrategy`, `DynamicLeverageRatioStrategy`, and `GoldfinchConfig` are deployed with a proxy-implementation pattern. Two high-severity issues were raised with these three contracts where the initialize function can be front-run (TOB-GFH-001), to configure the system maliciously and a lack of contract existence check (TOB-GFH-005). Additionally, `slither-check-upgradeability` is not included in a continuous integration pipeline.
Function Composition	**Moderate.** The functions across the Goldfinch codebase are complicated and nested. While some of the functions have Natspec comments, some of them are potentially out of date (TOB-GFH-015) or missing. The nested intricacy of these functions make it difficult to understand and test.

Front-Running	**Moderate.** While we reported a vulnerability around the initialization functions being front-run, we did not find additional attack vectors that affect the system architecture.
Key Management	**Weak.** We reported an issue related to plaintext secrets (owner private keys, Infura API keys, and Etherscan API keys) being used throughout the development codebase (TOB-GFH-002). The leakage of these keys would result in a loss of funds on the key on mainnet; or open up a denial-of-service attack vector for Infura.
Monitoring	**Moderate.** While some functions in the contracts emitted events, many admin setters failed to emit events (TOB-GFH-011). Additionally, we were not provided with an incident response plan or information on off-chain components in behavior monitoring.
Specification	**Weak.** We were provided with a Notion document, which had an outline of the `TranchedPool` and the expected high-level logic of the contract. However, this was insufficient documentation to understand their system architecture. While the whitepaper provided an outline and mathematical representation of 2 formulas, documenting the rest of the arithmetic logic in the system was also lacking. This made it significantly more difficult to determine what the code intended to do and verify its correctness.
Testing & Verification	**Weak.** While many components are tested, the `LeverageRatioStrategy` contracts are not tested; and tested functions are missing corner cases. The code does not have coverage enabled. The tests are also not run in a continuous integration pipeline, and there is no automated test generation.

Engagement Goals

The engagement was scoped to provide a security assessment of the Goldfinch V2 contracts.

Specifically, we sought to answer the following questions:

- Are there appropriate conditions set for all publicly callable functions?
- Could an attacker cause arbitrary state changes?
- Are there any arithmetic overflow or underflows affecting the code?
- Could an attacker manipulate the contracts by front-running transactions?
- Is it possible for participants to steal or lose tokens?

Coverage

`CreditLine` and related contracts. The `CreditLine` contracts store the terms of a loan for a Borrower, including the interest rate, payment period, late fee APR, credit limit, and other important values. Additionally, the contract will also keep track of the balance, distributions between interest and principal, and accrual amounts.

`DynamicLeverageRatioStrategy` and `FixedLeverageRatioStrategy`. The `LeverageRatioStrategy` contracts keep track of the expected leverage ratio to hold true between the junior and the senior tranches. We reviewed these contracts to verify that admins can update the expected leverage ratios and that callers will always receive the accurate leverage ratio for the system at a given time.

`GoldfinchConfig` and related contracts. The `GoldfinchConfig` stores addresses that are used across the Goldfinch system, allowing them to be shared efficiently across contracts. We assessed these contracts for correctness, using a combination of manual review and static analysis.

Periphery Contracts. The periphery contracts consisted of the `Borrower` contract (to be used by Borrowers) and the `TransferRestrictedVault` contract (to be used by Backers), meant to provide convenient wrapper functionality to allow users to interact with the core functionality more intuitively.

`SeniorPool`. The SeniorPool allows Liquidity Providers to deposit and withdraw funds into a pool more diversified than the junior tranche, allowing providers more interest and passive rewards through their deposits in Compound. The contract behaves as a large pool fund, and determines how much to invest into the junior and senior tranches, respectively.

StakingRewards. `StakingRewards` is used to earn GFI reward tokens by staking Fidu tokens, the tokens are vested for a period, if the reward tokens are withdrawn before vesting time is over, slashing is performed to adjust reward tokens. Users can also choose to lockup their tokens to earn a higher reward rate.

`TranchedPool` and related contracts. `TranchedPool` controls the deposits of the pool and allocates funds between the junior and the senior tranches of the pool. It also allows Backers to deposit funds directly into the junior tranche. These contracts were reviewed to ensure that the fund distribution was consistent with the Notion document provided, however, the documentation was a work-in-progress and modified during the engagement. We expect refinements to this process to continue after our audit.

Out of Scope

The following contracts were not considered during the scope of the engagement with IDEO CoLabs.

- Goldfinch V1
  - `CreditDesk.sol`
  - `Pool.sol`
  - `V2Migrator.sol`
- Any off-chain components – outlined further in Appendix D

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

❑ **Either use `hardhat-deploy` for these contracts with arguments for the `initialize` call or replace the use of the `initialize` with a `constructor`.** These actions will help remediate the front-running of the functions. TOB-GFH-001

❑ **Avoid hardcoding secrets and using the process environment to store secrets.** TOB-GFH-002

❑ **Ensure dependencies are up to date.** Several node modules have been documented as malicious because they execute malicious code when installing dependencies to projects. Keep modules current and verify their integrity after installation. TOB-GFH-003

❑ **Use neither `ABIEncoderV2` nor any other experimental Solidity feature.** Refactor the code such that structs do not need to be passed to or returned from functions. TOB-GFH-004

❑ **Implement a contract existence check before a `delegatecall`.** Document the fact that `suicide` and `selfdestruct` can lead to unexpected behavior, and prevent future upgrades from introducing these functions. TOB-GFH-005

❑ **Consider using the OpenZeppelin SafeERC20 library or otherwise adding explicit support for ERC20 tokens with incorrect return values.** TOB-GFH-006

❑ **There should be a single function to update `targetCapacity`, `maxRateAtPercent`, `minRateAtPercent`, `maxRate` and `minRate`, so all of the values can be updated in a single transaction.** TOB-GFH-007

❑ **Use the existing `DEFAULT_ADMIN_ROLE` instead of the `Owner` role.** TOB-GFH-008

❑ **Document the risks of not having a chainID opcode in the signature schema while using the USDC permit function.** This will ensure that users are aware of the risks associated with permits and post-deployment forks. TOB-GFH-009

❑ **Either:** TOB-GFH-010

❑ **Add events for all critical operations that result in state changes.** Events aid in TOB-GFH-011

❑ **Validate whether the `lockedUntil` period is passed for a given position before withdrawing it.** TOB-GFH-012

❑ **Restrict senior tranche deposits unless junior tranche has been locked.** TOB-GFH-012

❑ **Add zero-value checks for all function arguments to ensure that users cannot accidentally set incorrect values, misconfiguring the system.** TOB-GFH-014

❑ **Minimize the use of inheritance and helper functions in the codebase to ensure enhanced readability.** TOB-GFH-016

❑ **Ensure users are aware of Goldfinch's deployed contract address.** Additionally, closely analyze all aspects of the contract architecture, and identify the risks associated with an attacker deploying different versions. This will mitigate any phishing attacks on end-users. TOB-GLH-017

❑ **Either update the Natspec documentation to reflect the expectations around the Borrower contract or update V2Migrator to reflect an optional Borrower contract.** TOB-GLH-018

❑ **Review and properly document the above mentioned aspects of the codebase.** TOB-GLH-019

Long term

❑ **Carefully review the Solidity documentation, especially the "Warnings" section, as well as the pitfalls of using the delegatecall proxy pattern.** TOB-GFH-001

❑ **Use a hardware security module (HSM), which will ensure that the keys never leave the module.** TOB-GFH-002

❑ **Consider integrating automated dependency auditing into the development workflow.** If a dependency cannot be updated when a vulnerability is disclosed, ensure that the codebase does not use and is not affected by the vulnerable functionality of the dependency. TOB-GFH-003

❑ **Integrate static analysis tools like Slither into your CI pipeline to detect unsafe pragmas.** TOB-GFH-004

❑ **Carefully review the Solidity documentation, especially the "Warnings" section, and the pitfalls of using the `delegatecall` proxy pattern.** TOB-GFH-005

❑ **Beware the idiosyncrasies of ERC20 implementations, it has a history of misuses and issues.** Adhere to the token integration best practices outlined in the Token Integration Checklist in Appendix B. TOB-GFH-006

❑ **Analyze the effects of every function and document the post conditions that hold true to achieve consistent state throughout the system.** TOB-GFH-007

❑ **Create a process to ensure documentation stays up to date and document all roles, their relevant hierarchy and functionality.** TOB-GFH-008

❑ **Identify and document the risks associated with having forks of multiple chains, and identify mitigation strategies for scenarios.** TOB-GFH-009

❑ **Document best practices for Goldfinch users.** Among others, users must: TOB-GFH-010

❑ **Consider using a blockchain-monitoring system to track any suspicious behavior** TOB-GFH-011

❑ **Document and analyze all public functions' arguments to protect the code from issues caused by user-provided values.** TOB-GFH-012

❑ **Use Slither, which will catch functions that do not have zero-value checks.** TOB-GFH-014

❑ **Document the risks associated with privileged users and single points of failure.** Ensure that users are aware of all the risks associated with the system. TOB-GFH-015

❑ **Document the expected flow of contracts, how they intend to work together, and function call stacks to ensure they are easy to follow and understand.** TOB-GFH-016

❑ **Review the risks of third-party contract deployments on all aspects of the system to ensure that third-party interactions work as expected.** TOB-GLH-017

❑ **Always update the function-specific Natspec and documentation to reflect the expected execution of code, to ensure enhanced understanding and code readability.** TOB-GLH-018

❑ **Consider writing a formal specification of the protocol.** TOB-GLH-019

Findings Summary

#	Title	Type	Severity
1	Initialization function can be front-run	Configuration	High
2	Insufficient protection on sensitive keys	Configuration	Medium
3	Project dependencies contain vulnerabilities	Patching	Medium
4	ABIEncoderV2 is not production-ready	Patching	Undetermined
5	Lack of contract existence check on delegatecall will result in unexpected behavior	Data Validation	High
6	SafeERC20Transfer library is incompatible with non-standard ERC20 tokens	Data Validation	Low
7	Interdependent critical config values are being configured through multiple transactions	Configuration	Low
8	Use default Admin role instead of Owner role	Configuration	Informational
9	Lack of chain ID validation allows re-using signatures across forks	Authentication	Informational
10	Risks of using EIP2612	Configuration	Informational
11	Lack of events for critical operations	Error Reporting	Informational
12	Missing validation for lockup period of tokens on withdrawal	Data Validation	Low
13	Anyone can prevent the SeniorPool to invest in a TranchedPool	Data Validation	High
14	Lack of zero-value checks on functions	Data Validation	Low
15	Contract owner has too many privileges	Access Controls	Informational
16	Contract architecture is overcomplicated	Configuration	Informational

17	Allowing third-party deployed Borrower contract may result in unexpected behavior	Configuration	Informational
18	Borrower contract is not optional for V1 Migrations	Configuration	Informational
19	Lack of contract documentation makes codebase difficult to understand	Documentation	Informational

1. Initialization function can be front-run

Severity: High | Difficulty: High
Type: Configuration | Finding ID: TOB-GFH-001
Target: `DynamicLeverageRatioStrategy.sol, FixedLeverageRatioStrategy.sol, GoldfinchConfig.sol`

Description
The `DynamicLeverageRatioStrategy` contract has an initializer function that can be front-run, allowing an attacker to incorrectly initialize the contract.

Due to the use of the `delegatecall` proxy pattern, this contract cannot be initialized with its own constructor, and it has an initializer function:

```
function initialize(address owner) public initializer {
  require(owner != address(0), "Owner address cannot be empty");

  __BaseUpgradeablePausable__init(owner);

  _setupRole(LEVERAGE_RATIO_SETTER_ROLE, owner);

  _setRoleAdmin(LEVERAGE_RATIO_SETTER_ROLE, OWNER_ROLE);
}
```

Figure 1.1: `contracts/protocol/core/DynamicLeverageRatioStrategy.sol#L32-L40`

An attacker could front-run this function and initialize the contract with malicious values.

Exploit Scenario
Bob deploys the `DynamicLeverageRatioStrategy` contract. Eve front-runs the contract initialization and sets her own address for the `owner` address. As a result, she gains respective rights to update the pool with arbitrary leverage values.

Recommendations
Short term, either use `hardhat-deploy` for these contracts with arguments for the `initialize` call or replace the use of the `initialize` with a `constructor`. These actions will help remediate the front-running of the functions.

Long term, carefully review the Solidity documentation, especially the “Warnings” section, as well as the pitfalls of using the delegatecall proxy pattern.

2. Insufficient protection on sensitive keys

Severity: Medium
Type: Configuration
Target: `package.json, hardhat.config.js`

Difficulty: High
Finding ID: TOB-GFH-002

Description
Sensitive information such as Etherscan keys, API keys, and an owner test private key is stored in the process environment. This increases the likelihood of compromise, which would allow an attacker to steal funds from the protocol with owner privileges.

The following portion of the `hardhat.config.js` uses secrets directly from the process environment:

```
"verify": "npx hardhat etherscan-verify --api-key DQUC8Y678J5RN5P7XE9RT91SWI7SSEDD53
--solc-input --network",
```

Figure 2.1: `package.json#L17`

Moreover, if this repository is public in the future, the git commit history will retain these private keys. This will allow any attacker to use the API keys in these files and steal any funds being held by plaintext owner keys on mainnet.

Exploit Scenario
Alice, a member of the IDEO CoLabs team, has secrets stored in the process environment. Eve, an attacker, gains access to Alice's device and extracts the Infura key, allowing them to cause a denial of service attack on the front-end of the system. With the owner's private key, an attacker can steal the respective funds on mainnet.

Recommendation
We are still iterating over recommendations for this issue.

Short term, avoid hardcoding secrets and using the process environment to store secrets.

Long term, use a hardware security module (HSM), which will ensure that the keys never leave the module.

3. Project dependencies contain vulnerabilities

Severity: Medium
Type: Patching
Target: `package.json`

Difficulty: Low
Finding ID: TOB-GFH-003

Description
Although dependency scans did not yield a direct threat to the project under review, `yarn audit` identified dependencies with known vulnerabilities. Due to the sensitivity of the deployment code and its environment, it is important to ensure dependencies are not malicious. Problems with dependencies in the JavaScript community could have a significant effect on the repository under review. The below output details these issues.

NPM Advisory	Description	Dependency
1547	Signature Malleability	`elliptic`
1648	Use of a Broken or Risky Cryptographic Algorithm	`elliptic`
1674	Arbitrary Code Execution	`underscore`, `contract-proxy-kit`
1717	Uncontrolled Resource Consumption	`@openzeppelin/cli`
1770	Arbitrary File Creation/Overwrite due to insufficient absolute path sanitization	`tar`, `@opengsn/gsn`, `@nomiclabs/hardhat-truffle5`, `@openzeppelin/test-helpers`, `@openzeppelin/upgrades`, `@openzeppelin/cli`, `contract-proxy-kit`, `firebase-tools`, `truffle-hdwallet-provider`

Table 3.1: NPM advisories affecting project dependencies

Exploit Scenario
Alice installs the dependencies of the in-scope repository on a clean machine. Unbeknownst to Alice, a dependency of the project has become malicious or exploitable. Alice subsequently uses the dependency, disclosing sensitive information to an unknown actor.

Recommendations
Short term, ensure dependencies are up to date. Several node modules have been documented as malicious because they execute malicious code when installing dependencies to projects. Keep modules current and verify their integrity after installation.

Long term, consider integrating automated dependency auditing into the development workflow. If a dependency cannot be updated when a vulnerability is disclosed, ensure that the codebase does not use and is not affected by the vulnerable functionality of the dependency.

4. ABIEncoderV2 is not production-ready

Severity: Undetermined
Type: Patching
Target: `throughout`

Difficulty: Low
Finding ID: TOB-GFH-004

Description
The contracts use the new Solidity ABI encoder, `ABIEncoderV2`. This experimental encoder is not ready for production.

More than 3% of all GitHub issues for the Solidity compiler are related to experimental features, primarily `ABIEncoderV2`. Several issues and bug reports are still open and unresolved. `ABIEncoderV2` has been associated with more than 20 high-severity bugs, some of which are so recent that they have not yet been included in a Solidity release.

For example, in March 2019 a severe bug introduced in Solidity 0.5.5 was found in the encoder.

Exploit Scenario
The Core contracts are deployed. After the deployment, a bug is found in the encoder, which means that the contracts are broken and can all be exploited in the same way.

Recommendations
Short term, use neither `ABIEncoderV2` nor any other experimental Solidity feature. Refactor the code such that structs do not need to be passed to or returned from functions.

Long term, integrate static analysis tools like Slither into your CI pipeline to detect unsafe pragmas.

5. Lack of contract existence check on delegatecall will result in unexpected behavior

Severity: High
Type: Data Validation
Target: `Proxy.sol`, `GoldFinchFactory.sol`

Difficulty: High
Finding ID: TOB-GFH-005

Description
The `Proxy` contract uses the `delegatecall` proxy pattern. If the implementation contract is incorrectly set or is self-destructed, the proxy may not detect failed executions.

The `Proxy` contract is being deployed by `hardhat-deploy` library, it implements a payable fallback function that is invoked when proxy calls are executed. This function does not have a contract existence check:

```
function _fallback() internal {
       // solhint-disable-next-line security/no-inline-assembly
    assembly {
        let implementationAddress :=
            sload(0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc)
        calldatacopy(0x0, 0x0, calldatasize())
        let success := delegatecall(gas(), implementationAddress, 0x0, calldatasize(), 0, 0)
        let retSz := returndatasize()
        returndatacopy(0, 0, retSz)
        switch success
           case 0 {
               revert(0, retSz)
           }
           default {
              return(0, retSz)
           }
    }
}
```

Figure 5.1: `Proxy.sol#L22-L38`

Minimal proxies are being deployed to use `delegatecall` proxy pattern for `Borrower`, `TranchedPool` and `CreditLine` contracts, the minimal proxy implementation also lacks the contract existence check:

```
function deployMinimal(address _logic) internal returns (address proxy) {
```

```
    bytes20 targetBytes = bytes20(_logic);
    // solhint-disable-next-line no-inline-assembly
    assembly {
      let clone := mload(0x40)
      mstore(clone, 0x3d602d80600a3d3981f3363d3d373d3d3d363d730000000000000000000000000000)
      mstore(add(clone, 0x14), targetBytes)
      mstore(add(clone, 0x28),
0x5af43d82803e903d91602b57fd5bf3000000000000000000000000000000000000)
      proxy := create(0, clone, 0x37)
    }
    return proxy;
  }
```

Figure 5.2: `GoldfinchFactory.sol#L130-L141`

As a result, a `delegatecall` to a destructed contract will return success as part of the EVM specification. The Solidity documentation includes the following warning:

> *The low-level functions call, delegatecall and staticcall return true as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed.*

Figure 5.3: A snippet of the Solidity documentation detailing unexpected behavior related to `delegatecall`.

The proxy will not throw an error if its implementation is incorrectly set or self-destructed. It will instead return success even though no code was executed.

Exploit Scenario
Eve upgrades the proxy to point to an incorrect new implementation. As a result, each `delegatecall` returns success without changing the state or executing code. Eve uses this failing to scam users.

Recommendations
Short term, implement a contract existence check before a `delegatecall`. Document the fact that `suicide` and `selfdestruct` can lead to unexpected behavior, and prevent future upgrades from introducing these functions.

Long term, carefully review the Solidity documentation, especially the “Warnings” section, and the pitfalls of using the `delegatecall` proxy pattern.

References
- Contract Upgrade Anti-Patterns
- Breaking Aave Upgradeability

6. SafeERC20Transfer library is incompatible with non-standard ERC20 tokens

Severity: Low
Type: Data Validation
Target: `contracts/library/SafeERC20Transfer.sol`

Difficulty: Medium
Finding ID: TOB-GFH-006

Description
The `SafeERC20Transfer` library is meant to work with any ERC20 token. However, several high profile ERC20 tokens do not correctly implement the ERC20 standard.

The ERC20 standard defines, among others, two transfer functions:

- `transfer(address _to, uint256 _value) public returns (bool success)`
- `transferFrom(address _from, address _to, uint256 _value) public returns (bool success)`

However, several high profile ERC20 tokens do not return a boolean on one or both of these two functions. Starting from Solidity 0.4.22, the return data size of external calls is checked. As a result, any call to `transfer` or `transferFrom` will fail for ERC20 tokens that have an incorrect return value.

Some ERC20 tokens that will not work include assets with large market cap value, such as BNB, OMG and Oyster Pearl.

Exploit Scenario
Bob uses `SafeERC20Transfer` with an non-compliant ERC20 token that does not return boolean then, all the calls to `transfer` and `transferFrom` will unexpectedly revert.

Recommendations
Short term, consider using the OpenZeppelin SafeERC20 library or otherwise adding explicit support for ERC20 tokens with incorrect return values.

Long term, beware the idiosyncrasies of ERC20 implementations, it has a history of misuses and issues. Adhere to the token integration best practices outlined in the Token Integration Checklist in Appendix B.

References

- Missing return value bug - At least 130 tokens affected
- Explaining unexpected reverts starting with Solidity 0.4.22

7. Interdependent critical config values are being configured through multiple transactions

Severity: Low | Difficulty: High
Type: Configuration | Finding ID: TOB-GFH-007
Target: `contracts/staking/StakingRewards.sol`

Description
Reward rate calculation is dependent on `targetCapacity`, `maxRateAtPercent`, `minRateAtPercent`, `maxRate` and `minRate`. A total of five transactions will be required to update all of these values. Transactions are not guaranteed to be confirmed in the same or consecutive blocks, it is possible for a transaction to get confirmed hours later in case of network congestion. In such cases, depending upon the sequence of transactions; reward rate might not be expected for either previous or new configuration.

```
function setTargetCapacity(uint256 _targetCapacity) public onlyAdmin updateReward(0) {
  targetCapacity = _targetCapacity;
}

function setMaxRateAtPercent(uint256 _maxRateAtPercent) public onlyAdmin updateReward(0) {
  maxRateAtPercent = _maxRateAtPercent;
}

function setMinRateAtPercent(uint256 _minRateAtPercent) public onlyAdmin updateReward(0) {
  minRateAtPercent = _minRateAtPercent;
}

function setMaxRate(uint256 _maxRate) public onlyAdmin updateReward(0) {
  maxRate = _maxRate;
}

function setMinRate(uint256 _minRate) public onlyAdmin updateReward(0) {
  minRate = _minRate;
}
```

Figure 7.1: `contracts/staking/StakingRewards.sol#L515-L533`

Exploit Scenario
Eve, the admin of `StakingReward` contract, wants to update the reward rate calculation and have news values for `targetCapacity`, `maxRateAtPercent`, `minRateAtPercent`, `maxRate` and `minRate`. Eve sends the transactions, `setMaxRateAtPercent` was first to be included in the block. Due to network congestion the other transactions are not confirmed yet. If the

value of `maxRateAtPercent` is higher than current `minRateAtPercent`. Until remaining transactions are not confirmed, the reward rate will be zero or not expected.

Recommendations
Short term, there should be a single function to update `targetCapacity`, `maxRateAtPercent`, `minRateAtPercent`, `maxRate` and `minRate`, so all of the values can be updated in a single transaction.

Long term, analyze the effects of every function and document the post conditions that hold true to achieve consistent state throughout the system.

8. Use default Admin role instead of Owner role

Severity: Informational
Type: Configuration
Target: `throughout`

Difficulty: High
Finding ID: TOB-GFH-008

Description
The contracts use the OpenZeppelin's `AccessControl` contract to implement hierarchical role-based access control mechanisms. `Owner` role is created and it acts as admin for other roles in the respective contract. `Owner` role can be removed in favour of `DEFAULT_ADMIN_ROLE(0x00)`, since the `DEFAULT_ADMIN_ROLE` acts as admin for all existing and future roles by default. The `Owner` role is similar to `DEFAULT_ADMIN_ROLE`, but using `Owner` role adds a gas cost overhead of approximately ~20k gas per role per contract.

```
function __initialize__(
  address owner,
  string calldata name,
  string calldata symbol,
  GoldfinchConfig _config
) external initializer {
   [...]
  _setupRole(MINTER_ROLE, owner);
  _setupRole(PAUSER_ROLE, owner);
  _setupRole(OWNER_ROLE, owner);

  _setRoleAdmin(MINTER_ROLE, OWNER_ROLE);
  _setRoleAdmin(PAUSER_ROLE, OWNER_ROLE);
  _setRoleAdmin(OWNER_ROLE, OWNER_ROLE);
}
```

Figure 8.1: contracts/staking/GFI.sol#L24-L47

Recommendations
Short term, use the existing `DEFAULT_ADMIN_ROLE` instead of the `Owner` role.

Long term, create a process to ensure documentation stays up to date and document all roles, their relevant hierarchy and functionality.

9. Lack of chain ID validation allows re-using signatures across forks

Severity: Informational | Difficulty: High
Type: Authentication | Finding ID: TOB-GFH-009
Target: `contracts/protocol/core/SeniorPool.sol`

Description
The Goldfinch v2 Protocol token contracts uses EIP 2612 to provide EIP 712-signed approvals through a `permit` function when interacting with USDC. A domain separator is included as part of the signature scheme which includes the chainID. However, this chainID is fixed at deployment time. In the event of a chain hardfork post-deployment, the chainID cannot be updated and signatures may be replayed across both versions of the chain. As a result, an attacker can reuse signatures to receive user funds on both chains. The chainID can be included explicitly in the schema of the signature passed to permit to mitigate this issue and avoid having to re-generate the entire domain separator.

The `depositWithPermit` function using USDC's `permit`:

```
  /**
   * @notice Identical to deposit, except it allows for a passed up signature to permit
   *  the Senior Pool to move funds on behalf of the user, all within one transaction.
   * @param amount The amount of USDC to deposit
   * @param v secp256k1 signature component
   * @param r secp256k1 signature component
   * @param s secp256k1 signature component
   */
  function depositWithPermit(
    uint256 amount,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
  ) public override returns (uint256 depositShares) {
    IERC20Permit(config.usdcAddress()).permit(msg.sender, address(this), amount, deadline,
v, r, s);
    return deposit(amount);
  }
```

Figure 9.1: `contracts/protocol/core/SeniorPool.sol#L78-L95`

In particular, the `makeDomainSeparator` function in the USDC contract, where the chain ID is used, is only called at initialization:

```
    /**
     * @notice Initialize v2
     * @param newName   New token name
     */
    function initializeV2(string calldata newName) external {
        // solhint-disable-next-line reason-string
        require(initialized && _initializedVersion == 0);
        name = newName;
        DOMAIN_SEPARATOR = EIP712.makeDomainSeparator(newName, "2");
        _initializedVersion = 1;
    }

    /**
     * @notice Make EIP712 domain separator
     * @param name      Contract name
     * @param version   Contract version
     * @return Domain separator
     */
    function makeDomainSeparator(string memory name, string memory version)
        internal
        view
        returns (bytes32)
    {
        uint256 chainId;
        assembly {
            chainId := chainid()
        }
        return
            keccak256(
                abi.encode(
                    // keccak256("EIP712Domain(string name,string version,uint256
chainId,address verifyingContract)")
                    0x8b73c3c69bb8fe3d512ecc4cf759cc79239f7b179b0ffacaa9a75d522b39400f,
                    keccak256(bytes(name)),
                    keccak256(bytes(version)),
                    chainId,
                    address(this)
                )
            );
    }
```

Figure 9.1: USDC Contract Code - `makeDomainSeparator`

The signature schema does not account for the contract's chain. As a result, if a fork of Ethereum is made after the contract's creation, every signature will be usable in both forks.

Exploit Scenario
Bob holds $1,000 USDC worth of tokens on Eth1.0. Bob has submitted a signature to permit the Goldfinch contract to spend these tokens on his behalf. With Eth2.0, a fork on mainnet is executed. As a result, Eve is now able to reuse Bob's signature to transfer funds on the new chain and the old chain.

Recommendation
Short term, document the risks of not having a chainID opcode in the signature schema while using the USDC permit function. This will ensure that users are aware of the risks associated with permits and post-deployment forks.

Long term, identify and document the risks associated with having forks of multiple chains, and identify mitigation strategies for scenarios.

10. Risks of using EIP2612

Severity: Informational | Difficulty: High
Type: Configuration | Finding ID: TOB-GFH-010
Target: `contracts/protocol/core/SeniorPool.sol`

Description
The usage of EIP 2612 increases the risk of the permit function being front-run and phishing attacks to occur.

EIP 2612 introduces the use of signatures to replace the traditional `approve` and `transferFrom` flow. These allow a third party to transfer tokens on behalf of a user, with the verification of a signed message.

Firstly, the `permit` function can be front-run by an external party by submitting the signature first:

```
    /**
     * @dev Sets `value` as the allowance of `spender` over `owner`'s tokens,
     * given `owner`'s signed approval.
     *
     * IMPORTANT: The same issues {IERC20-approve} has related to transaction
     * ordering also apply here.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     * - `deadline` must be a timestamp in the future.
     * - `v`, `r` and `s` must be a valid `secp256k1` signature from `owner`
     * over the EIP712-formatted function arguments.
     * - the signature must use ``owner``'s current nonce (see {nonces}).
     *
     * For more information on the signature format, see the
     * https://eips.ethereum.org/EIPS/eip-2612#specification[relevant EIP
     * section].
     */
    function permit(address owner, address spender, uint256 value, uint256 deadline, uint8
v, bytes32 r, bytes32 s) external
```

Figure 10.1: `@openzeppelin/contracts/drafts/IERC20Permit.sol`

The implications of using EIP 2612 introduces a potential for a different party to front-run the initial caller's transaction. This would result in the intended caller's transaction failing (as the signature has already been used & the funds approved). This may have effects on external contracts that rely on a successful `permit()` to execute.

Secondly, an attacker can more easily steal a user's tokens by running a phishing campaign asking for signatures in a context that is unrelated to the contracts. The hash message may look benign and random to users:

```
function recover(
    bytes32 domainSeparator,
    uint8 v,
    bytes32 r,
    bytes32 s,
    bytes memory typeHashAndData
) internal pure returns (address) {
    bytes32 digest = keccak256(
        abi.encodePacked(
            "\x19\x01",
            domainSeparator,
            keccak256(typeHashAndData)
        )
    );
    return ECRecover.recover(digest, v, r, s);
}
```

Figure 10.2: USDC Contract Code

Exploit Scenario
Bob holds $1,000 USDC worth of tokens on Eth1.0. Bob has submitted a signature to permit the Goldfinch contract to spend these tokens on his behalf. With Eth2.0, a fork on mainnet is executed. As a result, Eve is now able to reuse Bob's signature to transfer funds on the new chain and the old chain.

Recommendations
Short term, either:
- Document the risks of `permit` being front-run and ensure external contracts and scripts are aware of this possibility
- Prevent the likelihood of a successful phishing attempt by adding user documentation and on-chain mitigations

Long term, document best practices for Goldfinch users. Among others, users must:
- Be extremely careful when signing a message

- Avoid signing messages from suspicious sources
- Always require hashing schemes to be public

References

- EIP 2612's Security Implications

11. Lack of events for critical operations

Severity: Informational
Type: Error Reporting
Target: `throughout`

Difficulty: Low
Finding ID: TOB-GFH-011

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

For instance, `setTargetCapacity` is used to change `targetCapacity`, which affects the reward calculation in the contract, it does not emit an event providing confirmation of that operation to the users:

```
function setTargetCapacity(uint256 _targetCapacity) public onlyAdmin updateReward(0) {
    targetCapacity = _targetCapacity;
}
```

Figure 11.1: `contracts/staking/StakingRewards.sol#L515-L517`

The following critical operations would also benefit from triggering events:

- `updateGoldfinchConfig`
- `StakingReward.setVestingSchedule`
- `StakingReward.setLeverageMultiplier`
- `StakingReward.setMinRate`
- `StakingReward.setMaxRate`
- `StakingReward.setMinRateAtPercent`
- `StakingReward.setMaxRateAtPercent`
- `TranchedPool._lockPool`

Without events, users and blockchain-monitoring systems cannot easily detect suspicious behavior or change in sophisticated configured value.

Recommendation
Short term, add events for all critical operations that result in state changes. Events aid in contract monitoring and the detection of suspicious behavior.

Long term, consider using a blockchain-monitoring system to track any suspicious behavior in the contracts. The system relies on several contracts to behave as expected. A monitoring mechanism for critical events would quickly detect any compromised system components.

12. Missing validation for lockup period of tokens on withdrawal

Severity: Low | Difficulty: Low
Type: Data Validation | Finding ID: TOB-GFH-012
Target: `contracts/protocol/periphery/TransferRestrictedVault.sol`

Description
In `TransferRestrictedVault`, positions are locked at the time of deposit for a certain period. The owner of the position cannot transfer the positions before the lock period is over. However, the position's funds can be withdrawn through the `withdrawJunior` function before the `lockedUntil` for the position has passed.

```
  function withdrawJunior(uint256 tokenId, uint256 amount)
    public
    nonReentrant
    onlyTokenOwner(tokenId)
    returns (uint256 interestWithdrawn, uint256 principalWithdrawn)
  {
    PoolTokenPosition storage position = poolTokenPositions[tokenId];
    require(position.lockedUntil > 0, "Position is empty");

    IPoolTokens poolTokens = config.getPoolTokens();
    uint256 poolTokenId = position.tokenId;
    IPoolTokens.TokenInfo memory tokenInfo = poolTokens.getTokenInfo(poolTokenId);
    ITranchedPool pool = ITranchedPool(tokenInfo.pool);

    (interestWithdrawn, principalWithdrawn) = pool.withdraw(poolTokenId, amount);
    uint256 totalWithdrawn = interestWithdrawn.add(principalWithdrawn);
    safeERC20Transfer(config.getUSDC(), msg.sender, totalWithdrawn);
    return (interestWithdrawn, principalWithdrawn);
  }
```

Figure 12.1: contracts/protocol/periphery/TransferRestrictedVault.sol#L160-L178

The funds for a respective position can be withdrawn through `withdrawSenior` and `withdrawSeniorInFidu` functions too before the `lockedUntil` for a position is passed.

Exploit Scenario
Bob deposits 100 USDC in the junior tranche through `depositJunior` function, the position is locked for 15 days. He withdraws the position after a few seconds regardless of `lockedUntil` time and 100 USDC are immediately transferred back to Bob.

Recommendation
Short term, validate whether the `lockedUntil` period is passed for a given position before withdrawing it.

Long term, document and analyze all public functions' arguments to protect the code from issues caused by user-provided values.

13. Anyone can prevent the SeniorPool to invest in a TranchedPool

Severity: High
Type: Data Validation
Target: `contracts/protocol/core/SeniorPool.sol`

Difficulty: High
Finding ID: TOB-GFH-013

Description
`SeniorPool` can invest in the junior tranche only if there is no prior deposit in the senior tranche of a Tranched pool. `SeniorPool` computes the appropriate investment amount for the senior tranche on the basis of leverage ratio and total amount deposited in the junior tranche of a Tranched pool.

```
  function investJunior(ITranchedPool pool, uint256 amount) public override whenNotPaused
nonReentrant onlyAdmin {
    require(validPool(pool), "Pool must be valid");

    // We don't intend to support allowing the senior pool to invest in the junior tranche
if it
    // has already invested in the senior tranche, so we prohibit that here. Note though
that we
    // don't care to prohibit the inverse order, of the senior pool investing in the senior
    // tranche after investing in the junior tranche.
    ITranchedPool.TrancheInfo memory seniorTranche =
pool.getTranche(uint256(ITranchedPool.Tranches.Senior));
    require(
       seniorTranche.principalDeposited == 0,
      "SeniorPool cannot invest in junior tranche of tranched pool with non-empty senior
tranche."
    );
   [...]
  }
```

Figure 13.1: `contracts/protocol/core/SeniorPool.sol#L202-L226`

If a very low amount is invested in the senior tranche (1 wei worth of USDC token), then the Senior pool will not be able to deposit into the junior tranche. To invest more in the senior tranche, more deposits in the junior tranche of a Tranched will be needed. This is a very low-cost griefing attack.

Exploit Scenario
Alice, admin of `SeniorPool` wants to invest 100 USDC in the junior tranche, Bob, notices the transaction and front-run it by depositing 1 decimal of usdc token in the senior tranche of that `TranchedPool` contract. Alice transaction reverts, she can not invest in junior tranche through `SeniorPool` contract, and until enough funds are not deposited to junior tranche, Alice can not invest in senior tranche too.

Recommendation
Short term, restrict senior tranche deposits unless junior tranche has been locked.

Long term, analyze and document the effects of locking the tranches of the pool on the rest of the system architecture.

14. Lack of zero-value checks on functions

Severity: Low
Type: Data Validation
Target: `throughout the codebase`

Difficulty: High
Finding ID: TOB-GFH-014

Description
Certain setter functions fail to validate incoming arguments, so callers can accidentally set important state variables to the zero address.

For example, the `initialize` function in the `Borrower` contract sets the addresses of the `admin` and the DAI, USDC, and whitelisted tokens:

```
function initialize(address owner, address _config) external override initializer {
    require(owner != address(0) && _config != address(0), "Owner and config addresses cannot
be empty");
    __BaseUpgradeablePausable__init(owner);
    config = GoldfinchConfig(_config);

    trustedForwarder = config.trustedForwarderAddress();

    // Handle default approvals. Pool, and OneInch for maximum amounts
    address oneInch = config.oneInchAddress();
    IERC20withDec usdc = config.getUSDC();
    usdc.approve(oneInch, uint256(-1));
    bytes memory data = abi.encodeWithSignature("approve(address,uint256)", oneInch,
uint256(-1));
    invoke(USDT_ADDRESS, data);
    invoke(BUSD_ADDRESS, data);
    invoke(GUSD_ADDRESS, data);
    invoke(DAI_ADDRESS, data);
  }
```

Figure 14.1: contracts/protocol/periphery/Borrower.sol#L38-L54

Once these addresses have been set to the zero address, they cannot be changed, and the `initialize` function can no longer be invoked. Changing these state variables would require the deployment of a new implementation contract.

This issue is also present in the following functions:

- `Borrower`
  - `initialize - trustedForwarder, oneInch`
- `GoldfinchConfig`
  - `setAddress - newAddress`

        - `setNumber - default value check missing`
        - `setTreasuryReserve - newTreasuryReserve`
        - `setSeniorPoolStrategy - newStrategy`
        - `setCreditLineImplementation - newAddress`
        - `setBorrowerImplementation - newAddress`
        - `setGoldfinchConfig - newAddress`
        - `addToGoList - _member`
    - `updateGoldfinchConfig - throughout codebase`

Exploit Scenario
Alice, a member of the Goldfinch team, creates a `Borrower` contract for a user, where the trustedForwarder address in the configuration is set to `address(0)`. As a result, the Borrower contract needs to be re-deployed with the correct `trustForwarder` address set.

Recommendations
Short term, add zero-value checks for all function arguments to ensure that users cannot accidentally set incorrect values, misconfiguring the system.

Long term, use Slither, which will catch functions that do not have zero-value checks.

15. Contract owner has too many privileges

Severity: Informational
Type: Access Controls
Target: throughout the code

Difficulty: Medium
Finding ID: TOB-GFH-015

Description
The owner of the contracts has too many privileges relative to standard users. Users are encompassed by Borrowers, Backers, and Liquidity Providers – however, there are high centralization risks in this codebase.

The contract owner can do the following:

- Upgrade the contracts at any point in time
- Set Up a malicious implementation of `TrustForward` contract and can arbitrarily mutate the `msg.value` and `msg.data`.
- Determine the distribution of funds between the amount of money going into the junior and senior tranche
- Sophisticated config values e.g. `GoldfinchConfig`, `DrawdownPeriodInSeconds`, `LatenessGracePeriodInDays`, `ReserveDenominator` etc. can be modified by `GoldFinchConfig's` admin

The concentration of these privileges creates a single point of failure.

Recommendation
Short term:

- Clearly document the functions and implementations the owner can change.
- Split privileges to ensure that no one address has excessive ownership of the system.

Long term, document the risks associated with privileged users and single points of failure. Ensure that users are aware of all the risks associated with the system.

16. Contract architecture is overcomplicated

Severity: Informational
Type: Configuration
Target: `throughout`

Difficulty: Low
Finding ID: TOB-GFH-016

Description
The contract relies on a significant usage of low-level manipulations for storage solutions and has split up functions into many nested calls to execute arithmetic calculations.

For example, the `collectInterestAndPrincipal` function, an internal function, in the TranchedPool makes 12 calls to helper functions in the contract to retrieve values, transfer funds, and calculate additional helper values:

```
  function collectInterestAndPrincipal(
    address from,
    uint256 interest,
    uint256 principal
  ) internal returns (uint256 totalReserveAmount) {
    safeERC20TransferFrom(config.getUSDC(), from, address(this), principal.add(interest),
"Failed to collect payment");

    (uint256 interestAccrued, uint256 principalAccrued) = getTotalInterestAndPrincipal();
    uint256 reserveFeePercent = ONE_HUNDRED.div(config.getReserveDenominator()); // Convert
the denonminator to percent

    uint256 interestRemaining = interest;
    uint256 principalRemaining = principal;

    // First determine the expected share price for the senior tranche. This is the gross
amount the senior
    // tranche should receive.
    uint256 expectedInterestSharePrice = calculateExpectedSharePrice(interestAccrued,
seniorTranche);
    uint256 expectedPrincipalSharePrice = calculateExpectedSharePrice(principalAccrued,
seniorTranche);

    // Deduct the junior fee and the protocol reserve
    uint256 desiredNetInterestSharePrice = scaleByFraction(
      expectedInterestSharePrice,
```

```
      ONE_HUNDRED.sub(juniorFeePercent.add(reserveFeePercent)),
      ONE_HUNDRED
    );
    // Collect protocol fee interest received (we've subtracted this from the senior portion
above)
    uint256 reserveDeduction = scaleByFraction(interestRemaining, reserveFeePercent,
ONE_HUNDRED);
    totalReserveAmount = totalReserveAmount.add(reserveDeduction); // protocol fee
    interestRemaining = interestRemaining.sub(reserveDeduction);

    // Apply the interest remaining so we get up to the netInterestSharePrice
    (interestRemaining, principalRemaining) = applyToTrancheBySharePrice(
      interestRemaining,
      principalRemaining,
      desiredNetInterestSharePrice,
      expectedPrincipalSharePrice,
      seniorTranche
    );

    // Then fill up the junior tranche with all the interest remaining, upto the principal
share price
    expectedInterestSharePrice = juniorTranche.interestSharePrice.add(
      usdcToSharePrice(interestRemaining, juniorTranche.principalDeposited)
    );
    expectedPrincipalSharePrice = calculateExpectedSharePrice(principalAccrued,
juniorTranche);
    (interestRemaining, principalRemaining) = applyToTrancheBySharePrice(
      interestRemaining,
      principalRemaining,
      expectedInterestSharePrice,
      expectedPrincipalSharePrice,
      juniorTranche
    );

    // All remaining interest and principal is applied towards the junior tranche as
interest
    interestRemaining = interestRemaining.add(principalRemaining);
    // Since any principal remaining is treated as interest (there is "extra" interest to be
distributed)
    // we need to make sure to collect the protocol fee on the additional interest (we only
```

```
deducted the
    // fee on the original interest portion)
    reserveDeduction = scaleByFraction(principalRemaining, reserveFeePercent, ONE_HUNDRED);
    totalReserveAmount = totalReserveAmount.add(reserveDeduction);
    interestRemaining = interestRemaining.sub(reserveDeduction);
    principalRemaining = 0;

    (interestRemaining, principalRemaining) = applyToTrancheByAmount(
      interestRemaining.add(principalRemaining),
      0,
      interestRemaining.add(principalRemaining),
      0,
      juniorTranche
    );

    sendToReserve(totalReserveAmount);

    return totalReserveAmount;
  }
```

Figure 16.1: `contracts/protocol/core/TranchedPool.sol#L550-L622`

Recommendation
Short term, minimize the use of helper functions in the codebase to ensure enhanced readability and ensure complex arithmetic operations are documented thoroughly.

Long term, document the expected flow of contracts, how they intend to work together, and function call stacks to ensure they are easy to follow and understand.

17. Allowing third-party deployed Borrower contract may result in unexpected behavior

Severity: Informational | Difficulty: High
Type: Configuration | Finding ID: TOB-GLH-017
Target: `contracts/protocol/periphery/Borrower.sol`

Description
Goldfinch's current architecture allows third parties to deploy different versions of the Borrower contract, which may result in unexpected behavior.

- Changing and updating the logic in the future
- Misconfigured contract, using the incorrect contract addresses for tokens
- Overriding the msg.sender value of transactions
- Malicious contract changes to `pay()` to fund an attacker's CreditLine instead of one submitted by a user
- Addition of a backdoor in the contract, allowing an attacker to steal funds from their Borrower contract

Recommendations
Short term, ensure users are aware of Goldfinch's deployed contract address. Additionally, closely analyze all aspects of the contract architecture, and identify the risks associated with an attacker deploying different versions. This will mitigate any phishing attacks on end-users.

Long term, review the risks of third-party contract deployments on all aspects of the system to ensure that third-party interactions work as expected.

18. Borrower contract is not optional for V1 Migrations

Severity: Informational | Difficulty: High
Type: Configuration | Finding ID: TOB-GFH-018
Target: `contracts/protocol/periphery/Borrower.sol`

Description
Goldfinch's current architecture allows third parties to deploy different versions of the Borrower contract:

```
 * @notice These contracts represent the a convenient way for a borrower to interact with
Goldfinch
  *  They are 100% optional. However, they let us add many sophisticated and convient
features for borrowers
  *  while still keeping our core protocol small and secure. We therefore expect most
borrowers will use them.
  *  This contract is the "official" borrower contract that will be maintained by Goldfinch
governance. However,
  *  in theory, anyone can fork or create their own version, or not use any contract at all.
The core functionality
  *  is completely agnostic to whether it is interacting with a contract or an externally
owned account (EOA).
```

Figure 18.1: `contracts/protocol/periphery/Borrower.sol#L18-L23`

However, if a Borrower contract does not exist for a V1 `CreditDesk`, the migration contract enforces that a Borrower contract exists for the user, otherwise the CreditDesk will not be migrated to a CreditLine:

```
  function migrateCreditLines(
    GoldfinchConfig newConfig,
    address[][] calldata creditLinesToMigrate,
    uint256[][] calldata migrationData
  ) external onlyAdmin {
    IBase creditDesk = IBase(newConfig.creditDeskAddress());
    IGoldfinchFactory factory = newConfig.getGoldfinchFactory();
    for (uint256 i = 0; i < creditLinesToMigrate.length; i++) {
      address[] calldata clData = creditLinesToMigrate[i];
      uint256[] calldata data = migrationData[i];
      address clAddress = clData[0];
      address owner = clData[1];
```

```
    address borrowerContract = borrowerContracts[owner];
    if (borrowerContract == address(0)) {
      borrowerContract = factory.createBorrower(owner);
      borrowerContracts[owner] = borrowerContract;
    }
    (address newCl, address pool) = creditDesk.migrateV1CreditLine(
      clAddress,
      borrowerContract,
      data[0],
      data[1],
      data[2],
      data[3],
      data[4],
      data[5]
    );
    emit CreditLineMigrated(owner, clAddress, newCl, pool);
  }
}
```

Figure 18.1: `contracts/protocol/periphery/V2Migrator.sol#L56-L60`

Recommendations
Short term, either update the Natspec documentation to reflect the expectations around the Borrower contract or update V2Migrator to reflect an optional Borrower contract.

Long term, always update the function-specific Natspec and documentation to reflect the expected execution of code, to ensure enhanced understanding and code readability.

19. Lack of contract documentation makes codebase difficult to understand

Severity: Informational
Type: Documentation
Target: `throughout`

Difficulty: Low
Finding ID: TOB-GFH-019

Description
The codebase lacks code documentation, high-level descriptions, and examples, making the contracts difficult to review and increasing the likelihood of user mistakes.

The codebase would benefit from detailed documentation, including on the following:

- A thorough explanation of migration flow from the V1 `CreditDesk` to the V2 `CreditLine` contracts
- The architectural overview of the flow of different contracts and how they intend to work together
- The junior and senior tranche distribution and in-depth derivations highlighting how the share price mutates through a variety of payments and drawdowns.
- The vocabulary used across the codebase, such as `writedown`, `drawdown`, leverage strategies, etc
- The areas of the whitepaper that have been implemented along with the respective contracts they pertain to in the codebase.
- Proper and detailed Natspec comments for the functions, highlighting the description about functionality and arguments in depth, especially for arithmetic functions.
- The optimization strategies used across the codebase, mapping to the storage slots and how they are used

The documentation should include all expected properties and assumptions relevant to the abovementioned aspects of the codebase.

Recommendations
Short term, review and properly document the above mentioned aspects of the codebase.

Long term, consider writing a formal specification of the protocol.

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 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
Timing	Related to race conditions, locking or 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 user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal

	implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order 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.
Centralization	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 purpose.
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 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 component was reviewed and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code Quality Recommendations

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

General Recommendations

- **Rename `ConfigOptions` to `GoldfinchConstants` to ensure the contract name accurately reflects what it does.**
- **Remove the unused `truffle-config.js` to ensure the codebase does not contain dead or unused code.**
- **Fix all Solidity Compiler warnings.**

```
contracts/protocol/core/LeverageRatioStrategy.sol:27:19: Warning: Unused function parameter.
Remove or comment out the variable name to silence this warning.
  function invest(ISeniorPool seniorPool, ITranchedPool pool) public view override returns
(uint256) {
                  ^--------------------^

contracts/protocol/core/LeverageRatioStrategy.sol:47:31: Warning: Unused function parameter.
Remove or comment out the variable name to silence this warning.
  function estimateInvestment(ISeniorPool seniorPool, ITranchedPool pool) public view
override returns (uint256) {
                               ^--------------------^

contracts/protocol/core/FixedLeverageRatioStrategy.sol:28:29: Warning: Unused function
parameter. Remove or comment out the variable name to silence this warning.
  function getLeverageRatio(ITranchedPool pool) public view override returns (uint256) {
                            ^----------------^

contracts/protocol/core/SeniorPool.sol:288:6: Warning: Unused local variable.
    (uint256 _, uint256 writedownAmount) = _calculateWritedown(pool, principalRemaining);
     ^-------^

contracts/protocol/periphery/TransferRestrictedVault.sol:183:5: Warning: Unused function
parameter. Remove or comment out the variable name to silence this warning.
    uint256 tokenId // solhint-disable-line no-unused-vars
    ^-------------^
```

Figure C.1: `Solidity Compiler Warnings`

`Vesting.sol`

- **Convert `uint256` data type of `startTime` and `endTime` to `uint64` to save 20,000 units of gas on each new instance of `Rewards`.**

```
uint256 startTime;
uint256 endTime;
```

Figure C.2: `contracts/library/Vesting.sol#L19-L20`

`CreditLine.sol`

- **Ensure that function names accurately reflect the execution of code.**
  - `CreditLine.handlePayment()`
  - `CreditLine.updateCreditLineAccounting()`

`Borrower.sol`

- **Use the inbuilt abi decoder (`abi.decode(_res,(uint256))`) to decode value instead of using low-level assembly code .**

```
function toUint256(bytes memory _bytes) internal pure returns (uint256 value) {
  assembly {
    value := mload(add(_bytes, 0x20))
  }
}
```

Figure C.3: `contracts/protocol/periphery/Borrower.sol#L274-L278`

D. Assumptions about Goldfinch

During the course of this review, a series of assumptions were provided regarding the functionality of Goldfinch. This code was not considered in scope for this code review. These assumptions include:

- **Goldfinch executes the due-diligence, screening and Know-Your-Customer (KYC) for Borrowers.** One key pillar in this architecture relies on Borrowers being trustworthy enough to pay back their loan within the terms set out in the `CreditLine` contract. The smart contracts reviewed prevent Borrowers from drawing down additional loans if they are late on payments, but do not penalize Borrowers for not repaying their loan.
- **Goldfinch accurately determines how much money to invest from the Senior Pool to the junior tranche.** Another key pillar in this architecture revolves around the deposits in the junior tranche being leveraged by 4x, into the senior tranche. The smart contracts do not calculate or verify the leverage ratio. This is assumed to be Goldfinch's responsibility to choose a safe ratio.
- **Goldfinch will make the final decision whether not to invest in the senior tranche of a loan or not.** If the leverage ratio between the junior and senior tranche makes the loan non-fundable, the SeniorPool should not invest at all. This ensures that the Borrower is unable to take out a loan of this amount.
- **The Unique Entity Check, meant to ensure that all actors in the system are unique, has not been implemented.**
- **The Auditor role, intended to screen Borrowers prior to entering the system, has not been implemented.**
- **The NFT redemption as defined in `PoolTokens.redeem()` is stable.** As a result, even if the resulting NFT received from a deposit has been redeemed and not burnt, the NFT will not be usable in this system.

E. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. An up-to-date version of the checklist can be found in crytic/building-secure-contracts.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken
```

To follow this checklist, use the below output from Slither for the token:

```
- slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
- slither [target] --print human-summary
- slither [target] --print contract-summary
- slither-prop . --contract ContractName # requires configuration, and use of Echidna and
Manticore
```

General Security Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on blockchain-security-contacts.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

ERC Conformity

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.
- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from

stealing tokens.

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Finally, there are certain characteristics that are difficult to identify automatically. Conduct a manual review of the following conditions:

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.
- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine if the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.

- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

F. Detecting Functions Missing `onlyAdmin` Modifiers in CreditLine

The Goldfinch codebase has many functions that are meant to be callable only by owners. As a result, most of the functions are protected by an `onlyAdmin` modifier. We used Slither to identify and review functions that are not protected by this modifier.

The script (included below) follows an approach in which every reachable function must be either protected by an `onlyAdmin` modifier or included on the `ACCEPTED_LIST`.

```
from slither import Slither
from slither.core.declarations import Contract
from typing import List

slither = Slither(".", ignore_compile=True)

def find_contract_in_compilation_units(contract_name: str) -> Contract:
    contracts = slither.get_contract_from_name(contract_name)
    return (contracts[0]) if len(contracts)>0 else print("Contract not found")

def _check_access_controls(
    contract: Contract, modifiers_access_controls: List[str], ACCEPTED_LIST: List[str]
):
    print(f"### Check {contract} calls {modifiers_access_controls[0]}")
    no_bug_found = True
    for function in contract.functions_entry_points:
        if function.is_constructor:
            continue
        if function.view:
            continue

        if not function.modifiers or (
            not any((str(x) in modifiers_access_controls) for x in function.modifiers)
        ):
            if not function.name in ACCEPTED_LIST:
                print(f"\t- {function.canonical_name} should have a
{modifiers_access_controls[0]} modifier")
                no_bug_found = False
    if no_bug_found:
        print("\t- No bug found")


_check_access_controls(
    find_contract_in_compilation_units("CreditLine"),
    ["onlyAdmin"],
    ["__BaseUpgradeablePausable__init", "__PauserPausable__init", "pause", "unpause",
"grantRole", "revokeRole", "renounceRole", "initialize"]
)
```

F.1: check-only-owner.py