Goldfinch Security Review

Auditors

Denis Milicevic, Lead Security Researcher

Optimum, Lead Security Researcher

tchkvsky, Junior Security Researcher

Report prepared by: Pablo Misirov and tchkvksy

February 3, 2023

Contents

1 About Spearbit

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

Learn more about us at spearbit.com

2 Introduction

Goldfinch is a decentralized, globally accessible credit protocol, with a mission to bring the world's credit activity on-chain while expanding access to capital and fostering financial inclusion. The protocol makes crypto loans without requiring crypto collateral—the missing piece that finally unlocks access to cryptocurrency capital for most people in the world. By incorporating the principle of trust through consensus Goldfinch creates a way for borrowers to show creditworthiness based on the collective assessment of other participants, rather than based on over-collateralizing with crypto assets.

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

3 Risk classification

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

3.1 Impact

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

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

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

4 Executive Summary

Goldfinch engaged with Spearbit for a seven-week period, with each week consisting of five engineering days spread out over a six-month retainer to review warbler-labs/mono protocol.

In this period of time a total of **87** issues were found.

Note that each review focuses on a specific portion of the codebase and is not meant to provide a comprehensive evaluation of the entire codebase. The portion reviewed is determined by the client and encompasses the areas deemed to be the riskiest. The findings and recommendations presented in each review period are based solely on the codebase portion reviewed and may not reflect the overall quality or security of the entire codebase.

Summary

Project Name	Goldfinch
Repository	warbler-labs/mono
Type of Project	Credit Protocol, DeFi
Audit Timeline	June 2022 - December 2022
Week 1	May 23rd - May 27th (Counted as June)
Week 2	July 4th - July 8th
Week 3	August 8th - August 12th
Week 4	September 19th - September 23rd
Week 5	October 24th - October 28th (Extended)
Week 6	November 21st - November 25th
Week 7	December 19th - December 23rd

Issues Found

Severity	**Count**
Critical Risk	1
High Risk	3
Medium Risk	10
Low Risk	16
Gas Optimizations	14
Informational	43
Total	**87**

5 Findings

6 May

Accounting for June's period, conducted between May 23rd - May 27th on commit hash c15f502b with the following files in scope: `StakingRewards.sol`, `PoolTokens.sol`, `flat_StakingRewards.sol` and `flat_PoolTokens.sol`

6.1 Medium Risk

6.1.1 Position seller might trick a buyer to buy a depreciated token

Severity: *Medium*

Description: Positions are implemented as ERC721, and therefore can be traded on decentralized exchanges. A position seller can spot the actual swap transaction in the mem-pool and front-run it with a transaction that depreciates the position value (e.g. calling `unstake`, `getReward`, etc), causing the buyer to buy a depreciated token. In the worst case, liquidity can be totally emptied from the position, since unstaking the entire amount does not cause the burn of the token.

Recommendation: The issue is caused due to the fact that the `tokenId` is not bonded to changes in the position storage state. We propose to implement a mechanism where there are two different identifiers for any token, an internal identifier that should stay constant throughout the entire token life-cycle (it can be implemented as a counter), and an external identifier. Whenever the position storage data is changed, the old token should be burned, and a new token should be minted using a new external identifier. The external identifier might be implemented as `hash(internal_id, version)`, where the version is incremented whenever the position storage data is changed.

6.1.2 Locked OpenZeppelin dependency 3.0.0 contains memory leaks that causes increased costs and could prevent execution

Severity: *Medium Risk* / *Gas Optimization*

Context: ERC721.sol#L137 ~ 20 references to this throughout the codebase All instances of SafeMath `sub div mod` are affected `sub` ~140 references `div` ~100 references `mod` ~15 references

Description: The Goldfinch contracts have a locked dependency of @openzeppelin/`contracts-upgradeable@3.0.0`. Versions prior to 3.4 contain memory leaks across various parts, such as its Enumerable types and SafeMath operators. What this means is that each subsequent call of one of these affected methods results in a quadratically increasing gas cost of it. In the best case, this results in end-users unnecessarily paying an increased gas overhead for transacting with the contracts. In the worst case, this could bubble up the gas usage to the point where the contracts can't execute.

Recommendation: It is recommended to upgrade the dependency to the latest available minor version, and generally an attempt should be made at each future deployment to check if there is a newer compatible minor version and inspect its associated changelog for any fixes that may affect the current set of contracts. It is duly noted that upgrading to the next major version, 4.0, is not possible safely with the current Goldfinch deployment, due to it targeting solc 0.6.x and also there existing storage incompatibilities between 3.0 and 4.0 series of OZ upgradables.

The recommended upgrade path is towards the latest @openzeppelin/`contracts-upgradeable@3.4.2` following Goldfinch's own due diligence, as it has some vendored parts of those contracts.

The upgrade should resolve all of the SafeMath operator issues, where custom error messages are not utilized. This seems to be the case for the contracts inspected within the scope of this audit. The memory leaking functions are still exposed but noted as deprecated with instructions on how to work around them in the case where custom error messages are utilized.

In the case of the `ownerOf` leak, it will require an override and dropping the deprecated `.get` custom error method with a `.tryGet` and appropriately handling failure conditions as reverts within it.

Albeit the upgrade itself, should at least minimize the risks of bubbling into a situation where the gas limit is reached, as the majority of memory leaking references were SafeMath-based.

6.2 Low Risk

6.2.1 Add postcondition to ensure `interestRedeemed` does not exceed a safe maximum

Severity: *Low Risk*

Context: PoolTokens.sol#L146

Description: The prior modified token properties have postconditions that ensure they are within a safe maximum. Such a postcondition for `interestRedeemed` appears to be missing, which depending on outside logic could cause more than expected or allowable interested to be pulled.

Recommendation: Additionally have a check that ensures the interest being redeemed is not above the safely allowable maximum for that specific token.

6.2.2 Missing sanity checks on royalty input and output parameters

Severity: *Low Risk*

Context: ConfigurableRoyaltyStandard.sol#L42-L56 PoolTokens.sol#L266-L283

Description: The setter and getter for the royalty is missing a variety of sanity checks to protect end-users. The setters within the scope of the audit appear to be access controlled via an `onlyAdmin` modifier, therefore it is a privileged function and the risk is low. However, a malicious admin could still steal the entire sale price, with the currently missing preconditions. Additionally, the admin could potentially front-run an expensive sale with an MEV sandwich attack where the maximally allowed royalty of a marketplace is charged, before the sale transaction, and then it is reset to a sane value following the sale transaction. In the worst case of a marketplace erroneously handling sales, the current lack of checks could even allow the admin to exceed sale price royalties and dip into a marketplace balance.

Recommendation: Add sanity checks ensuring a threshold royalty fee cannot be exceeded to provide some guarantee to end-users on what is deployed. Additionally consider a timelock mechanism before the new fee amount takes effect, to protect users from such hypothetical privileged attacks.

6.2.3 Missing null address check on royalty receiver

Severity: *Low Risk*

Context: ConfigurableRoyaltyStandard.sol#L55

Description: Allows the setting of a royalty, but the royalty receiver is not validated and could end up being a null address, which could result in accidental scenarios where it is omitted while a royalty fee is set, and the royalty does not reach the intended royalty receiver.

Recommendation: Have an appropriate precondition via a require statement to ensure the parameter is not being set to null address alongside a non-zero fee.

6.3 Gas Optimization

6.3.1 Two same div ops can be reduced to one to save gas

Severity: *Gas Optimization*

Context: StakingRewards.sol#L335

Description: Following the multiplication, the result is twice divided by `MULTIPLIER_DECIMALS`. This results in an unnecessary extra division operation, as it's being divided by the same constant twice.

Recommendation: Cut the operations down to a single SafeMath `div` call, which uses a constant that is the square of `MULTIPLIER_DECIMALS`.

6.3.2 Redundant null address check

Severity: *Gas Optimization*

Context: PoolTokens.sol#L134-L135

Description: The condition at L135 would basically ensure that the `token.pool` & `poolAddr` are not a null address, as the `msg.sender`, which this implementation of `_msgSender()` seems to fallback on, should never be able to be the null address.

Recommendation: The null address check on L134 is therefore redundant and can be removed, as long as the code on L135 is not modified from its current implementation.

6.3.3 Redundant ERC165 interface check already covered by EIP173Proxy

Severity: *Gas Optimization*

Context: PoolTokens.sol#L24 PoolTokens.sol#L289 EIP173Proxy.sol#L34

Description: The `PoolTokens` contract is deployed behind a proxy that has its own `supportsInterface` implementation, which does basic checks according to the standard, including ERC165 interface support. Any additional logic handling is then passed to the implementation contract for its own `supportsInterface`. The implementation contract here, unnecessarily checks for ERC165 support once again.

In the case ERC165 support is checked, that condition is unreachable, as it'll be preemptively hit in the EIP173Proxy. In the case ERC2981 support is checked for, or any other unsupported interface, this yields an additional unnecessary condition check that will never be true.

Recommendation: Remove the unnecessary ERC165 interface definition in the `PoolTokens` contract and the check for it in the `supportsInterface` function within.

6.3.4 Redundant non-existent token checks in dependency

Severity: *Gas Optimization*

Context: ERC721.sol#L397-L399 ERC721.sol#L269-L270

Description: In the case the spender is not the owner, a redundant non-existent token check occurs that is not necessary via the `getApprove` call.

Recommendation: Since this is in an open-source dependency of the project, it be best to notify the maintaining team but also consider contributing a fix, whereby there is a `getApproved` internal variant available for this callchain, that does not redundantly recheck token existence, or even just replaces the `getApproved` call with `_tokenApprovals[tokenId]` as that is all the function would contain anyways without the require.

6.4 Informational

6.4.1 Redundant calls to `updateReward(0)`

Severity: *Informational* **Context:** StakingRewards.sol#L383 StakingRewards.sol#L392 StakingRewards.sol#L446

Description: The functions mentioned call `updateReward(0)`, while in practice it is redundant since the same lines of code are executed twice.

Recommendation: Consider removing the call to `updateReward(0)`.

6.4.2 Call to `_additionalRewardsPerTokenSinceLastUpdate` during `updateReward` always yields 0

Severity: *Informational* / *Gas Optimization*

Context: StakingRewards.sol#L933-L944 StakingRewards.sol#L233-L242 StakingRewards.sol#L227-L231 StakingRewards.sol#L200-L225

Description: During the callchain to `updateReward(tokenId)`, with a non-zero parameter, `_additionalRewardsPerTokenSinceLastUpdate(block.timestamp)` is called via L944 -> L240 -> L230 to L212. It depends on the `lastUpdateTime`, which has already been set to the current `block.timestamp` in the lines prior to L944. Meaning there might be some issues in the logic or that the call to at L230 is dead code.

Recommendation: Ensure the logic is currently correct above all and that `lastUpdateTime` should indeed be set before L944.

In the case that it is, supplementing L230 with a preemptive return when `block.timestamp`and `lastUpdateTime` are equal.

6.4.3 Replace `updateReward` modifier with inlined calls for consistency

Severity: *Informational* / *Gas Optimization*

Context: File.sol#L123

Description: This modifier pattern is introduced but is not consistently usable across the codebase. In certain cases, the function is directly called, as it is not possible to call it only before or after all logic, but somewhere in between. This has introduced an ambiguity in its use and introduced cases also where there are multiple redundant calls to the same function, leading to a waste of gas, inefficient EVM usage, and extra gas cost to end-users.

Recommendation: Remove the `updateReward` modifier, and inline the calls where necessary.

6.4.4 Fully annotate all public interfaces using NatSpec

Severity: *Informational*

Context: PoolTokens.sol#L208-L214

Description: There are multiple instances of public interfaces (anything that makes it into the ABI) that are missing supporting NatSpec annotations. This hurts code readability, understanding and maintenance.

Recommendation: Appropriately annotate all the public interfaces using NatSpec.

6.4.5 Document all known and expected behaviours

Severity: *Informational*

Context: PoolTokens.sol#L224-L231

Description: In addition to returning, the function in question reverts when the `tokenId` does not exist. This is not noted and leads to auditors or anyone else inspecting the code having to dig deeper into each function to understand what all the possible behaviours and effects are. In turn, this can introduce development overhead in the best case, where the documentation does not fully cover associated effects or in the worst case an assumption that it doesn't revert and build a future feature that fails as it lacked that assumption.

Recommendation: Attempt to document all known and expected effects using `@dev` natspec tag, where it can be concisely communicated within the documentation or at least state there are additional exceptions.

6.4.6 Set internally unused public visibility to external for keeping consistent with external/internal pattern

Severity: *Informational*

Context: PoolTokens.sol#L233 StakingRewards.sol#L614-L627

Description: The contracts in scope authored by Goldfinch generally follow the external/internal implementation pattern, whereby an external function calls into an internal variant prefixed with an underscore that implements the actual logic.

The functions in context, such as `validPool`, unnecessarily breaks from this pattern with the public visibility specifier, which is unnecessary, as it does not appear to be accessed internally, and could lead to unexpected issue, if the public and internal variant were to deviate in some future iteration and the internal variant used internally in certain parts, and the public elsewhere.

Recommendation: Stay consistent with the external/internal pattern and make the function external only.

6.4.7 Naming clash between declared variable and struct property "pool"

Severity: *Informational*

Context: PoolTokens.sol#L238-L243

Description: A variable named `pool` is declared at the beginning of the function scope. Within this function scope, the `TokenInfo` `pool` property is accessed as well. This naming clash can cause confusion and ambiguity. In the case of Visual Studio Code with Solidity and auditor functions, it erroneously states that the `pool` property on L243 is the variable declared on L238.

Recommendation: Name the declared variable something different that still keeps its intent clear.

6.4.8 Check and upgrade to latest compatible dependencies when possible instead of vendoring them

Severity: *Informational*

Context: PR#474

Description: The greatest portion of code additions from this PR is the vendoring of a number of dependency contracts for one function that was missing the virtual keyword to be overridable. This dependency is currently locked to 3.0.0.

Recommendation: The upstream dependency since version 3.4.0 has the necessary changes implemented and should be compatible with the codebase, meeting the needs to run on solc 0.6.x and with the current set of utilized 3.0.0 OZ upgradeable contracts. It is recommended to upgrade to the latest 3.4.2 release, and revert the then unnecessitated vendored dependencies from that PR.

In the future considering checking the latest compatible version, before vendoring dependencies. And even in the case vendoring is needed, for such cases as adding virtual keywords, there may be other projects and good

reason to implement it for the entire dependency, therefore considering contributing the change to the dependency itself.

7 July

Conducted between July 4th - July 18th on commit hash 00845f86 with the following files in scope: `TranchedPool.sol`, `ImplementationRepository.sol`, `UcuProxy.sol`, `VersionedImplementation.sol`, `StakingRewards.sol` and `SafeMath.sol`

7.1 Medium Risk

7.1.1 `SafeERC20Transfer` - Not compatible with nonconforming ERC20 implementations

Severity: *Medium Risk*

Context: SafeERC20Transfer.sol#L22, SafeERC20Transfer.sol#L42, SafeERC20Transfer.sol#L61

Description: The custom `SafeERC20Transfer` used performs a check to enforce a `true` return value from a non-reverting call. This mitigates problems with contracts like the old MiniMe implementation, where failed transfers do not revert and simply return `false`.

There are, however, other nonconforming ERC20 implementations that will revert at all times if called through the `SafeERC20Transfer` reviewed. Any ERC20 token not returning a boolean will fail explaining-unexpected-reverts-starting-with-solidity-0-4-22.

The current USDT token is one example of a token which does not return a boolean on transfers.

Custom SafeERC20Transfer lib:

```
bool success = erc20.transfer(to, amount);
require(success, message);
```

Comparing to the latest OZ lib (safeTransfer and _callOptionalReturn), where the USDT no return value is handled.

Recommendation: Make use of the OpenZeppelin `SafeERC20` library over the custom `SafeERC20Transfer` library. There is a version in the existing OpenZeppelin dependencies.

7.1.2 `UcuProxy.upgradeImplementation` - Possible front running issues

Severity: *Medium Risk*

Context: Description: In order to upgrade the implementation of a `UcuProxy`, a new implementation needs to be added to `ImplementationRepository` by calling `ImplementationRepository.append` first, then (optionally), the upgrade data should be determined by calling `ImplementationRepository.setUpgradeDataFor`, and finally, `UcuProxy.upgradeImplementation` should be called to launch the actual upgrade.

In case the calls to `ImplementationRepository.setUpgradeDataFor`, `UcuProxy.upgradeImplementation` are executed in a non-atomic way (i.e. not within a single transaction), the upgrade process is susceptible to miner/validator front running attack vector, where the miner can abandon the `ImplementationRepository.setUpgradeDataFor` transaction, or process it after the call to `UcuProxy.upgradeImplementation`, therefore causing an upgrade without initialization.

In addition, the owner of `ImplementationRepository` can front run a call to `UcuProxy.upgradeImplementation` changing `nextImplementation` and/or the proposed upgrade data.

Recommendation: As for the first issue, consider removing `ImplementationRepository.setUpgradeDataFor` and including its logic in `ImplementationRepository.append` instead.

As for the second issue, consider adding the expected values for the implementation and the upgrade data to `UcuProxy.upgradeImplementation`, or alternatively, introduce a timelock mechanism to limit the potential timing of changes to the current implementation.

7.2 Low Risk

7.2.1 `ImplementationRepository._remove` - violates assumptions about upgradeability

Severity: *Low Risk*

Context:

- Assumption 1: `upgradeDataFor` assumes `TranchedPool` is upgrading from a particular version ImplementationRepository.sol#L197
- Assumption 2: `TranchedPool`s can be upgraded if there is a new implementation in the same lineage ImplementationRepository.sol#L199

Description: Assumption 1 is violated when `remove`ing. In most cases there is no impact, unless `upgradeDataFor` depends on the previous implementation being a particular version.

Assumption 2 is violated when `remove`ing due to the statement: `_nextImplementationOf[toRemove] = INVALID_IMPL;`. When the proxy attempts to upgrade *from* a removed implementation, `ImplementationRepository.nextImplementationOf` returns `address(0)` restricting an upgrade from proceeding.

Marked as low risk as the `ImplementationRepository` itself is upgradeable.

Recommendation: For assumption 1, consider:

- creating a checklist to review before appending or removing an implementation to confirm these problems are not present
- ensuring all implementations in a lineage are compatible with each other and there is no version dependency on a particular previous version when using `upgradeDataFor`

For assumption 2, consider:

- leaving `_nextImplementationOf[toRemove]` untouched for the removed implementation, provided there is no conflict in upgrading each version to any other future version. NOTE: this runs the risk of the version being upgraded to later being removed which is acceptable as `_nextImplementationOf[toRemove]` is not deleted.

7.2.2 Solidity version no longer pinned to `pragma solidity 0.6.12`

Severity: *Low Risk*

Context: IERC173.sol#L3, IVersioned.sol#L4, SafeMath.sol#L1GoldfinchFactory.sol#L3, TranchedPoolImplementationRepository.sol#L3, ImplementationRepository.sol#L3, VersionedImplementationRepository.sol#L3, UcuProxy.sol#L3

Description: Solidity version no longer pinned to `pragma solidity 0.6.12` and instead uses `pragma solidity >=0.6.12`

Recommendation: For production releases use a pinned solidity version.

7.3 Gas Optimization

7.3.1 `ImplementationRepository._append` - Multiple reads to `_currentOfLineage[lineageId]`

Severity: *Informational* / *Gas Optimization*

Context: ImplementationRepository.sol#L171-L183

Description: Multiple reads to `_currentOfLineage[lineageId]`.

Recommendation: Can save a tiny bit of gas by using the cached read:

```
/// @notice Set an implementation to the current implementation
/// @param implementation implementation to set as current implementation
/// @param lineageId id of lineage to append to
function _append(address implementation, uint256 lineageId) internal virtual {
  require(Address.isContract(implementation), "not a contract");
  require(!_has(implementation), "exists");
  require(_lineageExists(lineageId), "invalid lineageId");
+
+  address oldImplementation = _currentOfLineage[lineageId];
-  require(_currentOfLineage[lineageId] != INVALID_IMPL, "empty lineage");
+  require(oldImplementation != INVALID_IMPL, "empty lineage");

-  address oldImplementation = _currentOfLineage[lineageId];
  _currentOfLineage[lineageId] = implementation;
  lineageIdOf[implementation] = lineageId;
  _nextImplementationOf[oldImplementation] = implementation;

  emit Added(lineageId, implementation, oldImplementation);
}
```

7.4 Informational

7.4.1 `StakingRewards` - unused functions

Severity: *Informational*

Context: StakingRewards.sol#L140, StakingRewards.sol#L931, StakingRewards.sol#L59

Description: `ZAPPER_ROLE` no longer has special permission meaning the zapper functions are unneeded.

Recommendation: Remove the two unused functions and one unused constant.

7.4.2 `VersionedImplementationRepository` - Pack version tightly

Severity: *Informational* / *Gas Optimization*

Context: VersionedImplementationRepository.sol#L21, VersionedImplementationRepository.sol#L54, VersionedImplementationRepository.sol#L60, VersionedImplementationRepository.sol#L65

Description: Packing the `uint8` array packs into `bytes` 3 words wide (i.e. `abi.encodePacked([1,0,9])` packs into `0x0000000000000000000000000000000000000000000000000000000000000001000000000000000000000000000000000000000000000009`).

Packing tighter saves some gas: `abi.encode(version[0], version[1], version[2])` packs into `0x010009`.

Recommendation: Modify each instance of `abi.encodePacked(version)` (and `abi.encode(version)`) to `abi.encode(version[0], version[1], version[2])`.

```
function getByVersion(uint8[3] calldata version) external view returns (address) {
+  return _byVersion[abi.encodePacked(version[0], version[1], version[2])];
-  return _byVersion[abi.encodePacked(version)];
}

...snip...

function _insertVersion(uint8[3] memory version, address impl) internal {
  require(!_hasVersion(version), "exists");
+  _byVersion[abi.encodePacked(version[0], version[1], version[2])] = impl;
-  _byVersion[abi.encodePacked(version)] = impl;
  emit VersionAdded(version, impl);
}

function _removeVersion(uint8[3] memory version) internal {
+  bytes memory versionKey = abi.encode(version[0], version[1], version[2]);
+  address toRemove = _byVersion[versionKey];
-  address toRemove = _byVersion[abi.encode(version)];
+  _byVersion[versionKey] = INVALID_IMPL;
-  _byVersion[abi.encodePacked(version)] = INVALID_IMPL;
  emit VersionRemoved(version, toRemove);
}

function _hasVersion(uint8[3] memory version) internal view returns (bool) {
+  return _byVersion[abi.encodePacked(version[0], version[1], version[2])] != INVALID_IMPL;
-  return _byVersion[abi.encodePacked(version)] != INVALID_IMPL;
}
```

Modify comment VersionedImplementationRepository.sol#L12. `address` takes up a single slot, `bytes` are simply hashed to determine which storage slot to use.

7.4.3 `UcuProxy` - Imports different `Address` implementation

Severity: *Informational*

Context: UcuProxy.sol#L8

Description: `UcuProxy` imports a different Address than the one imported through `StakingRewards`

@openzeppelin/contracts/utils/Address.sol vs @openzeppelin/contracts-ethereum-package/contracts/utils/Address.sol

Recommendation: Consolidate on one of the two version, `@openzeppelin/contracts/utils/Address.sol` is the more up to date of the two.

7.4.4 Spellcheck on comments

Severity: *Informational*

Context: ImplementationRepository.sol#L10, TranchedPool.sol#L109 and others

Description: Some misspelled words in natspec comments.

Recommendation: Good practice to make use of a spellchecker IDE plugin.

7.4.5 `TranchedPool` - Centralization risk with the two transfers to admin controlled address

Severity: *Informational*

Context: TranchedPool.sol#L332

Description: Calling `emergencyShutdown` transfers funds to the `reserveAddress`. The current implementation restricts admin from resetting the `reserveAddress`, however, the logic in the GoldfinchConfig contract is upgradeable.

It is noted that GoldfinchConfig upgrades are currently controlled by a multisig.

Recommendation: Consider:

- documenting risk
- adding timelock to GoldfinchConfig upgrades

7.4.6 Mixed use of SafeMath for `numSlices +/- 1`

Severity: *Informational* / *Gas Optimization*

Context: TranchedPool.sol#L572, TranchedPool.sol#L285, TranchedPool.sol#L308, TranchedPool.sol#L593

Description: SafeMath is used in some places and not others for the same calculation.

Recommendation: Adopt a style guide to use SafeMath in a consistent way.

In instances where over/under flow is not possible, use unchecked arithmetic (default in the Solidity version used; 0.6.12) add a comment on why it is not possible to over/under flow.

7.4.7 `TranchedPool` - `safeERC20TransferFrom` used where `safeERC20Transfer` is sufficient

Severity: *Informational* / *Gas Optimization*

Context: TranchedPool.sol#L256, TranchedPool.sol#L487, TranchedPool.sol#L557. May exist in unreviewed contracts as well.

Description: The `TranchedPool` already has sufficient access to call `transfer` for tokens it owns; using `transferFrom` incurs additional gas costs as the `allowed mapping` is updated

Recommendation: When transferring from `address(this)` use a `safeTransfer` over `safeTransferFrom`

7.4.8 `TranchedPool` - `public` functions can be `external`

Severity: *Informational*

Context: TranchedPool.sol#L174, TranchedPool.sol#L137, TranchedPool.sol#L332, TranchedPool.sol#L355, TranchedPool.sol#L363, TranchedPool.sol#L377, TranchedPool.sol#L427, TranchedPool.sol#L453

Description: `public` functions not called by the contract can be `external`.

Recommendation: Edit the function visibility to `external`.

7.4.9 Inconsistent use of `_msgSender()` and `msg.sender`

Severity: *Informational*

Context: TranchedPool.sol#L123

Description: Throughout `msg.sender` is used over `_msgSender`.

Recommendation: Select a consistent convention. If not using GSN or related relay, consider simply `msg.sender` throughout.

7.4.10 `TranchedPool` - duplicate/unused SafeMath import

Severity: *Informational*

Context: TranchedPool.sol#L8

Description: `SafeMath` import line in `TranchedPool`. `SafeMath` is already imported and used though `BaseUpgradeablePausable`.

Recommendation: Remove unused import.

7.4.11 `TranchedPool` changes are backwards incompatible

Severity: *Informational*

Context: TranchedPool.sol#L41

Description: PR#779 contains breaking changes for `TranchedPool`, for instance, changing `poolSlices` to be a mapping instead of an array would cause the previous values stored in the array to be effectively lost.

Recommendation: Make sure that this version of code is not used to upgrade already deployed `TranchedPool` contracts, and rather used for new deployments only instead.

8 August

Conducted between August 8th - August 12th on commit hash e863eb4b with the following files in scope: `Accountant.sol`, `Go.sol` and `UniqueIdentity.sol`

8.1 Low Risk

8.1.1 `UniqueIdentity` - Missing withdrawal function for the token mint fee charged

Severity: *Low Risk*

Context: UniqueIdentity.sol#L96

Description: The `_mintTo` function charges a `MINT_COST_PER_TOKEN` fee to end-users to cover associated KYC costs. The contract thereby receives value, however, there is no withdrawal function implemented within the current version to allow this accruing balance to be accessed.

Recommendation: Implement a privileged withdraw function with supporting role, slated for a future contract version to access this balance.

8.1.2 `UniqueIdentity` - Potential front running / replay attack vectors

Severity: *Low Risk*

Context: UniqueIdentity.sol#L15

Description:

1. `UniqueIdentity` uses nonces to identify signatures uniquely but the structure of the signed data is identical for `mint` and `burn`. This effectively positions the signer as a trusted single point of failure. Let's consider the scenario where the signer has generated two different signatures meant for the `mint` function, for Alice (with nonces 0,1 respectively). Assuming the second signature was somehow leaked to Eve (the attacker), and that Alice has already used the first one to mint a UID, Eve can now use the second one to burn Alice's UID without her consent.
2. `UniqueIdentity#burn` can be called by anyone with a valid signature on behalf of someone else, which opens up a potential front-running issue. Let's say Alice (could be a contract/EOA) has a UID she wishes to burn. Alice is transmitting the `burn` transaction which gets front-runned (as is) by someone else, causing her transaction to fail (while the first transaction runs successfully), which (depends on Alice's client logic) may cause a false sense of failure for the entire burning process.

Recommendation:

1. Consider adding the function signature (also known as the 4 bytes identifier) to the hash preimage to distinguish between signatures meant for `mint` and `burn`.
2. The off-chain logic used for interaction with the `UniqueIdentity` contract should take the described scenario into consideration and handle this type of transaction failure properly.

8.2 Gas Optimization

8.2.1 Gas optimizations

Severity: *Gas Optimization*

Context:

1. UniqueIdentity.sol#L111-112

Description:

1. `UniqueIdentity#burn` - The check to validate that the balance has to be 0 after burning is redundant. Assuming that there are no accounts with an id balance > 1 caused by previous code versions, a user balance for a specific `id` can be either 0 or 1. The `_burn` call will either burn exactly 1 token or revert, and there's no external call that can cause unwanted behavior with reentrancy, thus `accountBalance` has to be 0, and the check is redundant.

8.3 Informational

8.3.1 Signer key leak risk in case of ECDSA signing nonce leak or nonce re-use by black box signer

Severity: *Informational*

Context: UniqueIdentity.sol#L138-L140

Description: The off-chain signer depends upon OpenZeppelin Defender which utilizes AWS KMS as a keystore and for cryptographic operations. Based on preliminary research, the HSM in use is certified but the ECDSA certifications explored appear to state they are simply conformance tests to ensure correct implementation of the algorithm but make no testatments necessarily to the security of the cryptographic processes.

In the case of a signing nonce leak (k), where the black box either predictably chooses a nonce, or leaks the one used with an accompanying signature, the private key used and stored could be obtained.

Similarly, if a signing nonce (k) is ever re-used for signing 2 different messages, the 2 resulting signatures could be used to obtain the signing key.

Recommendation: Using audited and open-source solutions is most ideal, where the signing nonce worries can be put to rest. The black box in question is not open-sourced, however, it is still among the best options in the market for teams not looking to have their own on-site key security logistics. The Goldfinch team should reach out to the service providers in question and ensure they have appropriate measures in place on the HSMs to avoid these potential risks, with most of both being solvable by RFC6979.

8.3.2 Carefully set and utilize long time constants depending on their application

Severity: *Informational*

Context: Accountant.sol#L29

Description: The line in question sets a `SECONDS_PER_YEAR` constant, which does not account for leap years, and would eventually go out of synchronicity with a calendar if used for such a purpose. The solidity `years` literal was removed in a previous version due to the confusion arising as to whether it should or should not account for leap years soliditylang.org.

Recommendation: In this case, a change is not recommended, as the `Accountant` contract appears to utilize a Actual/365 Basis for interest accrual, in which case using exactly 365 days based on 86400 second days without account for non-leap years should be fine. This informational issue is being brought up to be careful in regards to other potential uses of such time constants across the protocol so as to be careful whether leap years should be accounted for to keep synchronicity, and to ensure consistency of such constants when used for accounting.

8.3.3 `Go` - Prefer stricter relational operators where possible

Severity: *Informational*

Context: Go.sol#L98, Go.sol#L105

Description: The `goOnlyIdTypes` function queries the UID token balances of entities attempting to interact with it, but it accepts any balance greater than zero, while the only valid possible values are either 0 or 1 for the current implementation.

Recommendation: Utilize a stricter equality check of 1 instead, which signals a valid contained `UniqueIdentity`. This is both safer in the case of some unforeseen changes happening to `UniqueIdentity` and is more inline with the specification, where values greater than 1 would not be valid and should not be considered as such in the code.

8.3.4 `UniqueIdentity` - Unused digital signatures will be practically revoked once the signer was removed

Severity: *Informational*

Context: UniqueIdentity.sol#L140

Description: Signatures that were given through the RPC but were not used before the signer was removed will be practically revoked.

Recommendation: Make sure that the off-chain signer logic supports the scenario described above.

9 September

Conducted between September 19th - September 23rd on commit hash a9f35eaf with the following files in scope: `SimpleAssetVault.sol`, `GFIDirector.sol`, `Epochs.sol`, `ERC721NonTransferable.sol`, `CapitalDirector.sol`, `AccessControl.sol`, `Base.sol`, `ContextBuilder.sol`, `Router.sol`, `Routing.sol`, `Context.sol`, `GFILedger.sol`and `ICapitalAssets.sol`. Note that coverage does not extend to all the abovementioned contracts.

9.1 High Risk

9.1.1 `GFILedger` - Missing `GFI` tokens transfers

Severity: *High Risk*

Context: GFILedger.sol#L38-L43 GFILedger.sol#L45-L50

Description: The `GFILedger` contract appears to track and receive the GFI tokens used for the membership rewards. When depositing it is supposed to pull the deposited `GFI` tokens from `MembershipOrchestrator`to itself, and when withdrawing, it is supposed to send the `GFI` tokens back to the owner. The contract properly tracks the adding and removal of those tokens from its internal tracker, but it does not do any actual transfer of said tokens, effectively leaving any deposited tokens stuck in the contract with this implementation or effectively burned. There is an assert, but it will always hold, as the assert simply ensures that the actual balance of tokens within the contract is equal to or greater than the tracked balance, which is a symptom this issue will exactly create. There is additionally no indication that this contract approves another contract to handle these transfers on its behalf.

Recommendation: Amend logic within the `deposit` and `withdraw` functions to actually transfer the tokens to the contract and out of the contract back to owners respectively, or add logic to allow an operator or admin to approve another contract to do the transfers. The former is likely simpler and safer.

9.2 Medium Risk

9.2.1 Missing null address check on `AccessControl` admin component, where null address is default admin

Severity: *Medium Risk*

Context: AccessControl.sol#L65-L72

Description: With the current design of the `AccessControl` contract. The default admin of every unset contract, is `address(0)`. Under the case of directly utilizing `msg.sender`, this is unlikely to be an issue, as the null address shouldn't ever be `msg.sender`. However, under this architecture, it is operators setting this variable that is potentially forwarded here. Even a benign operator could issue a `tearDown()` call on the context, and reset the `msgSender` to be the null address, in which case passing that variable to this, on a newly deployed and unset contract could lead to admin-level exploitation in scenarios.

With the other potential issue of operators not having exclusive locks to the context and being able to inject `msgSender`, this could lead to critical level exploits across the architecture.

Even under solidity's own `msg.sender` it is bad practice to have the null address be the default programmatically.

Recommendation: Introduce a null address check to avoid `address(0)` being able to act as a default admin for all contracts in this architecture. Also consider a general reworking of the admin StateVar, and potentially just inheriting the operator methods for admin as well, as the current admin system is both limited and intricate, with any contract only being allowed a single set admin, while the operator system by default has all accounts unpermissioned, and allows permissioning of multiple accounts per contract which is overall better, but there is a risk of hanging permissions. Consider directly reading `msg.sender` as well if possible to reduce these potential attack surfaces.

9.2.2 `Context.setup` relies on a weak mutex that can lead to malicious variable injection

Severity: *Medium Risk*

Context: Context.sol#L34-L50 ContextBuilder.sol#L20-L24

Description: In the current case of the `withContext` modifier, any external call within the body that potentially leads to an operator that does a teardown, would lead to the `msgSender` context being set to `address(0)`. This may be then abused by this or another operator to set a different `msgSender` context. If the external calls then successfully finish, and return to the body here, and then the use of `msgSender` context follows within the body, it could lead to exploitable conditions via this malicious injection or variable clearing.

Likewise an exploited or malicious operator accessed could do the same and arbitrarily set this variable due to the weak mutex, which simply checks that it is `address(0)` but doesn't provide an exclusive lock.

The current mutex pattern simply depends on the last `msgSender` context being cleared before a new one can be set. The operations are doable by any operator, with one operator being able to clear a still dependent upon context of another.

Recommendation: An exclusive lock should be introduced, whereby the current lock can only be lifted by the address or contract that requested the lock in the first place. Precautions do need to be taken to avoid potential for deadlock. The GF team had communicated that the intent currently is for only one contract's context to be tracked here, so this should fit within their designs. Another option in the case they wish it to track multiple different contract contexts, is to track them via mappings, whereby other contracts or operators won't be able to maliciously alter the state of other ones.

9.2.3 `SimpleAssetVault._burnToken` may run out of gas for a user with many positions

Severity: *Medium Risk*

Context: SimpleAssetVault.sol#L248-L260

Description: `SimpleAssetVault._burnToken` may not be able to complete with today's mainnet gas limits once a user's balance exceeds ~5000 based on preliminary naive tests. Effectively, it means a user will not be able to withdraw any of his opened positions.

Recommendation: Since the purpose of this function is to remove a single `ownerToken`, then the index of the position can be provided by the function caller, and line 250 will be enough for validation. The `ownerTokens` array represents the tokens held by a specific `position.owner` there is no concrete potential for a front-running denial of service that can be caused by others relying on the fact that the index of a token might be changed in removals.

Let's explain why the potential front-running denial of service is not a concern here using an example. In case `ownerTokens = [a,b,c]`, and the owner is willing to remove `c`, then `_burnToken(c,2)` should be called. Since `c` is the last element, then in theory this transaction will revert if `_burnToken(b,1)` will front-run it. But, these operations are only callable by the position's owner.

9.2.4 Potentially malicious call via operator could lay false claim to unowned balances within `GFILedger`

Severity: *Medium Risk*

Context: GFILedger.sol#L38-L43

Description: Contracts within the cake framework can accept multiple operators. In the case of `GFILedger`, if a malicious operator were added, they could lay arbitrary and false claim when the condition `context.gfi().balanceOf(address(this)) > total` is true, allowing an operator to claim `context.gfi().balanceOf(address(this)) - total` to any arbitrary address. At the same time, a benign operator could use this as a rescue function for missent funds.

A malicious operator would generally lead to a complete security failure and is therefore considered low likelihood, however, in this case a compromised operator is not necessary, but potentially a malicious call via an operator. For this audit, the full call logic to this function is missing for audit, so this issue serves as a warning to ensure somewhere early in the call, there is an actual check for a deposit/transfer of tokens, otherwise anyone could potentially misuse the operator for this exploit, for them to lay claim to tokens they have not deposited. Within the current logic, no such check is available.

Alongside the currently missing withdrawal logic, this issue which on its own is considered medium risk, could be utilized with that other issue to create an essentially critical level exploit in the case of an upgradable contract. In that, users deposit their funds, and then withdraw. The ledger will untrack their balances, but never send them their tokens. Anytime before the update or after the update, assuming a metamorphic or proxy upgradable contract, and assuming no changes to the deposit logic, a malicious operator could lay claim to all those balances for themselves, and after update successfully withdraw the GFI tokens to their account.

Recommendation: The potential impact of this issue is greatly reduced with proper withdrawal logic. Additionally, the impact is also limited in the case of an immutable contract rather than an upgradable one. The attack surface of this issue is greatly reduced, if somewhere within the call in or before the call to `deposit`, it is ensured that the callee/msg.sender has indeed transferred the tokens being claimed as deposited, ideally it would be checked within this contract that actually holds the tokens.

9.3 Low Risk

9.3.1 Critical state-changing functions should be supplemented by events

Severity: *Low Risk* / *Informational*

Context: AccessControl.sol#L26-L54

Description: The `AccessControl` contract handles the auth for multiple contracts and has methods to add, remove, and set varying levels of permissions across these. In the current iteration, changes of state resulting from these are not accompanied by supporting events.

In the best case, this can make tracking and auditing of the state here, especially with a nested mapping, unnecessarily difficult for the team and also for users interested.

In the worst case, the team could leave hanging permissions from the lack of tracking which could introduce security risks within the architecture.

Having these events and the team actively logging and tracking them could also alert the team in case of any account compromises via a watchdog service, which has helped minimize exploit impacts in certain previous cases.

Recommendation: Supplement these critial state-changing functions with supporting events. An example recommended re-iteration of this contract, accounting only for the admin related methods during the audit was:

```
contract suggestedAC {
  mapping(address => address) public admins;
  event adminSet(address indexed resource, address indexed admin);
  error isNotAdmin(address resource, address accessor);
  constructor (address admin) {
    _setAdmin(address(this), admin);
  }
  function setAdmin(address resource, address admin) external {
    requireAdmin(address(this), msg.sender);
    _setAdmin(resource, admin);
  }
  function _setAdmin(address resource, address admin) private {
    admins[resource] = admin;
    emit adminSet(resource, admin);
  }
  function requireAdmin(address resource, address accessor) public view {
    bool isAdmin = admins[resource] == accessor;
    if (!isAdmin) revert isNotAdmin(resource, accessor);
  }
}
```

It includes a few other recommendations that came up during the audit, but not all, so should be taken as a pseudo example to build off of.

9.4 Gas Optimization

9.4.1 Caching variables in `memory` is cheaper than in `storage`

Severity: *Gas Optimization*

Context: SimpleAssetVault.sol#L244 SimpleAssetVault.sol#L245

Description: `position` and `ownerTokens` are called multiple times from `storage`. Using `memory` will cost less gas.

Recommendation: Cache a variable in `memory` if it is read multiple times. L251 and L252 will need to be amended in this case to directly access and set the `owners` variables.

9.4.2 Unnecessary repeated state writes that produce no side-effect and conversions in `SimpleAssetVault`

Severity: *Gas Optimization*

Context: SimpleAssetVault.sol#L203-L214 SimpleAssetVault.sol#L132

Description: The `_checkpoint` function repeatedly does a write to a struct member containing the timestamp the last checkpoint occurred, however, in most cases this has no side-effect on the contract whatsoever, essentially leading to sunken gas costs for saving a changing higher resolution variable (timestamp in seconds) that is converted into a lower resolution variable (timestamp in weeks) that is unchanged in most cases when actually used.

Additionally on the 2nd context, this leads to cases where unnecessary conversions and additional operations are introduced as overhead.

Recommendation: The first portion of the unnecessary state writes could be remedied by only saving the variable when a side-effect is imminent and expected to occur. This could be saving the timestamp only when it results in an epoch change.

The ideal change likely here, is just decreasing the time resolution to weeks instead of seconds, which is essentially what the Epoch representation is. It would both cut down on the unnecessary writes, unnecessary conversion at read times as you convert it just upon write, and it may give more headroom for struct packing if more members were to be introduced.

9.4.3 Tight-packing of struct possible in `SimpleAssetVault`

Severity: *Gas Optimization*

Context: SimpleAssetVault.sol#L59-L66

Description: The first member is an address which occupies 20 bytes or 160-bits, and the next 2 are currently set to 256-bits meaning all 3 members utilize 3 storage slots, whilst the first technically only occupies one partially. The 3rd member is a timestamp, and 256-bits is overkill for a seconds-based timestamp.

To give an idea, it could be packed with the address type, by setting it a uint96. Based on some naiive calculations this should provide a time resolution exceeding 200+ octodecillion years, which would be more than enough as it is approaching infinite times the age of the universe. Utilizing uint64 would give a time resolution exceeding 500+ billion years, or 42 times the current age of the universe.

Recommendation: Reduce the timestamp bitsize to any between 64 and 96 to capitalize on tightly-packed structs and utilize only 2 storage slots instead of 3. Additionally you will have to align the smaller bit members together for the compiler to take advantage of this.

9.5 Informational

9.5.1 `tokenByIndex()` will revert if `index` exceeds `totalSupply()`

Severity: *Low Risk* / *Informational*

Context: SimpleAssetVault.sol#L118-L120

Description: According to ERC721 Specification, `tokenByIndex()` will revert if `index` > `totalSupply()`. This also applies to `tokenOfOwnerByIndex()`.

Recommendation: Add a check or custom error to handle cases.

9.5.2 Comments and logic does not match

Severity: *Informational*

Context: SimpleAssetVault.sol#L92-L97 CapitalDirector.sol#L25-L28

Description: Comment shows contract is initialized and controlled by `owner` while logic does not show `owner` initialization.

Recommendation: Update comment to match logic.

9.5.3 Typo in `ICapitalAssets`

Severity: *Informational*

Context: ICapitalAssets.sol#L21

Description: A little typo in the comment of `getUsdcEquivalent()`

Recommendation: Change "assetW" to "asset"

9.5.4 Refactoring of `AccessControl` for improved consistency and quality

Severity: *Informational*

Context: AccessControl.sol#L51-L54

Description: Previous audits had commended the use of the separation of auth vs logic in external and private functions respectively as good practices. This contract is not following this pattern which has led to some missed opportunities for improved code quality and maintenance.

As an example, the constructor has its own isolated logic for adding an admin when it should ideally re-use functionality to set an admin.

Likewise there are 2 duplicated methods, `requiresAdmin` and `requiresSuperAdmin` with essentially the same logic. `requiresAdmin` can already do the limited scope of what `requiresSuperAdmin` does.

Therefore the latter should either be removed, as showcased in a previous refactoring example of this contract, or should just call and reuse `requiresAdmin`.

Recommendation: Consider applying the external/private function pattern followed in previous contracts, and removing any duplicate functions that already existing functions can handle the workload of, or at least ensuring their respective code blocks are not duplicated to improve code reuse and maintanability.

Please refer to the recommendation in issue Issue 26 for a suggested example of these changes and visualized improvements to the code.

9.5.5 Use a locked compiler version pragma

Severity: *Informational*

Context: Base.sol#L2

Description: An unlocked pragma may produce ambiguity as to the solc version to use for its compilation and on which versions it has actually been tested and is intended for. This can additionally produce friction between internal engineering, where different members could be testing and building for different versions.

Recommendation: Use a locked pragma to clearly signal the intended version to be used for the contracts.

Only a single contract within context is mentioned, however, this applies to all within scope of this audit.

9.5.6 Setting constructor visibility is obsolete since solc v0.7, replace with `abstract` where necessary

Severity: *Informational*

Context: Base.sol#L19

Description: Specification of constructor visibility was an initial method of setting a contract to be abstract or not (e.g. non-deployable or deployable). With the `abstract` keyword declaration available since solc 0.7, setting constructor visibility is considered obsolete.

Recommendation: Remove any visibility specifiers on the constructors of contracts within this audit scope, that are intended to be compiled with 0.8.16+, and replace cases with internal visibility to an abstract contract.

The context here mentions only a single contract as an example, but it is seen throughout the codebase and should be amended across all contracts.

9.5.7 Avoid unused imports

Severity: *Informational*

Context: Base.sol#L5

Description: The `Router` import appears unused within this scope.

Recommendation: Avoid having unnecessary and unused imports as a best practices measure.

9.5.8 `AccessControl` name clash between contract name and library constant

Severity: *Informational*

Context: Routing.sol#L22-L23

Description: The library's constant shares the name of a contract imported within that contract's scope. This can cause ambiguity and shadowing. Additionally the library is named `Contracts` which can cause further confusion to the expected type returned by that call.

Recommendation: Consider renaming the library to something more apt like RoutingKeys, or RoutingDict, and suffixing or prefixing Key or K to the current variable names, to avoid confusion and avoid potential instances of shadowing from that library and imports.

9.5.9 tx.* built-in symbol shadowed within Context by function

Severity: *Informational*

Context: Context.sol#L54

Description: The `tx()` function shadows the `tx.*` built-in, making it and its `tx.origin` and `tx.gasprice` properties inacessible in this contract and any that may inherit it. This could also be misused as an underhanded solidity coding technique, whereby the struct is expanded to include the origin and gasprice members, and the declared struct variable is named `tx` and set public, whereby it would transparently shadow `tx.*` and its properties within this contract and across any inheriting ones.

Recommendation: Avoid shadowing of any existing variables, especially solidity's globals and built-ins. Rename this function to something that doesn't clash with any other names, potentially `txnCtx` or if switching it to just `msgSender` access consider `userSender`.

9.5.10 Restrict visibility on all `Context` functions and set `view` specifier where applicable

Severity: *Informational*

Context: Context.sol#L37 Context.sol#L47 Context.sol#L54

Description: The visibility of these functions is currently set to public, however, they are never utilized within this contract, and this contract appears to be a standalone contract that is not inherited. In this case, it is limitable to external and will help clarify intent and limit the scope of the functions.

Additionally, the currently named `tx()` function on L54 only does a state read, and should have the `view` specifier added.

Recommendation: Limit their visibility to external. Add view specifier on L54 to currently named `tx()` function. Always setting these to the most constrained is recommended both to clarify their intent and limit potential attack surface.

9.5.11 A single member struct is unnecessary and inefficient

Severity: *Informational* / *Gas Optimization*

Context: Context.sol#L15-L17

Description: Structs are considered complex types used to create more complicated data types that contain multiple properties of potentially different types. This struct contains only a single property, deeming it potentially unnecessary and inefficient.

Recommendation: Consider just utilizing a single state variable address, which is all the struct contains anyways, it should lead to simpler and more efficient code.

10 October - November

Conducted between October 24th - October 28th and November 21st - November 25th on commit hash 7ea8714a targeting the following files in scope: `Membership Contracts`, `Cake Framework Contracts`, `SeniorPool.sol`, `ConfigHelper.sol`, `ConfigOptions.sol`, `TranchedPool.sol`, `WithdrawalRequestToken.sol`, `Zapper.sol`, `StakingRewards.sol`, `ISeniorPool.sol`, `ISeniorPoolEpochWithdrawals.sol`, `IUniqueIdentity0612.sol`and `IWithdrawalRequestToken.sol`. Note that coverage does not extend to all the abovementioned contracts.

10.1 Low Risk

10.1.1 `SeniorPool` - no threshold checks on configurable cancel fee & withdrawal fee could lead to malicious drains

Context: SeniorPool#L238 SeniorPool#L266

Description: During withdrawal cancellation or claiming, accompanying fees are charged depending on respective action. These are set in the configuartion and do not appear to have appropriate threshold checks in place, that would prevent the entirety of a user's amount to going to fees instead of withdrawal.

In general, this set of contracts with their configuration are dependent on a trusted, honest and competent operator, which so far the Goldfinch team appears to have done that role, it would however be ideal to minimize potential opportunities and impacts of a malicious operator.

Recommendation: Introduce appropriate threshold limits hardcoded into the contract, so the aforementioned configurable fees cannot be used maliciously to drain withdrawing user's balances via claim or cancel.

10.1.2 `SeniorPool` - initialization of a request's parameter adds value instead of setting it

Context: SeniorPool#L218

Description: Upon the initialization of a new request within `requestWithdrawal`, the intialization of `fiduRequested` is done by adding any previous value that may have been within that request to the `fiduAmount` argument. This is an anti-pattern for what is meant to be a clean intialization, and could lead to exploit conditions if some method for `_withdrawalRequest` collisions or decrementing of counters associated with them existed or were to be introduced.

Recommendation: In our audit analysis, this did not appear to be exploitable under the current implementation, however, it should still be just set to `fiduAmount`, which will save the unnecessary safeMath opcode costs and improve security by reducing the aforementioned attack surface.

10.1.3 `SeniorPool` - uint256 to int256 unchecked conversion and arithmetic could lead to overflow pre solidity 0.8.x

Context: SeniorPool#L623

Description: The contract directly typecasts 2 `uint256` types into 2 `int256`. As this contract is on solidity 0.6.12 this could render a silent overflow during conversion, if the `uint256` values exceed `type(uint256).max / 2`, leading to undesired results and potentially further overflow in the arithmetic.

Recommendation: Require the `uint256` values being typecast to be equal to or less than `type(uint256).max / 2` to yield values within the safe unsigned range of `int256` to avoid overflow on conversion and arithmetic.

10.1.4 `SeniorPool` - missing `SafeMath` when solidity 0.6.12

Context: SeniorPool.sol#L143 SeniorPool.sol#L743

Description: The contract in question is targeted for solidity version 0.6.12, this specific version does not check and revert for overflows or underflows by default. This can lead to unexpected states and behaviors leading to exploit, especially in the case of user-controlled or influenced input values.

In the case of L143, the value can be immediately overflowed by users. The reversion is only likely to occur due to inherent properties of the `USDC` contract, where the amount needed to overflow is unlikely to ever be available to a user, assuming proper function. Thereby it should fail at the `transferFrom` invocation. However, the origin contract should maximize its own defensive logic rather than depending on assumptions.

Recommendation: In just about all cases, except for increments by 1, it is a good idea for pre-0.8 solidity contracts to enforce safe math operations, to match the behavior of "modern solidity".

10.1.5 `SeniorPool` - `_applyWithdrawalRequestCheckpoint` may lock user funds in case an existing withdrawal request was not claimed for a long time

Context: SeniorPool.sol#L419

Description: USDC Withdrawals are implemented in a two-phase mechanism where the user has to first request a withdrawal and then (in a separate call) claim the USDC available. the call to `claimWithdrawalRequest` will iterate through all the epochs that were not claimed yet, and calculate the available amount of USDC that can be claimed. The for loop that's being used for that might cause the transaction to go out-of-gas, since it contains few storage operations, effectively causing a permanent denial of service, and thus the lock of the user's funds. Based on a pen to paper preliminary tests we have made, it is safe to assume that given the current epoch duration it's highly unlikely that a transaction will go out of gas, however, this issue is still possible and more likely to happen in case of a shorter epoch duration.

Recommendation: Consider adding a function that acts similarly to `_applyWithdrawalRequestCheckpoint` but iterates from `_withdrawalRequests[tokenId].epochCursor` to a specific index determined by the user (it has to be `<= _checkpointedEpochId`). This way, a user experiencing an out-of-gas exception will still be able to withdraw his funds in multiple transactions, thus avoiding the denial of service.

10.1.6 `SeniorPool.setEpochDuration` - Missing input validation for `newEpochDuration`

Context: SeniorPool.sol#L101-L112

Description: `setEpochDuration` is used by the admin of `SeniorPool` to set the duration of the current and future epochs. `setEpochDuration` does not validate that `newEpochDuration` is greater than zero, which might lead potentially to a denial of service caused by a division by zero inside `_mostRecentEndsAtAfter` . **Recommendation:** Consider reverting all calls to `setEpochDuration` with `newEpochDuration = 0`.

10.2 Gas Optimization

10.2.1 `SeniorPool` - events emitting a potentially unnecessary `address(0)` constant

Context: SeniorPool#L196 SeniorPool#L223 SeniorPool#L252

Description: These events are emitting a constant `address(0)` value for their kycAddress parameter.

Recommendation: Remove this unnecessary parameter and value since it is the same on every event emit and potentially obsolete.

10.2.2 `SeniorPool` - redundant setting of `request.epochCursor` after `_applyEpochAndRequestCheckpoints` already does it

Context: SeniorPool#L187

Description: The `epochCursor` for the request in question may be set twice to the same value in the same call. It appears to be redundant and unnecessary, as this second instance only occurs under the precondition that `fiduRequested == 0` for the request. But in all cases, the first instance of appropriately setting the `epochCursor` happens via `_applyEpochAndRequestCheckpoints` via L183 -> L436 -> L427.

Recommendation: Remove this redundant setting of the `epochCursor` and its accompanying precondition, which will yield some gas savings during run and deploy.

10.3 Informational

10.3.1 `SeniorPool.cancelWithdrawalRequest` - multiple invocations possible, triggering unnecessary 0-value transfers and event emissions

Context: SeniorPool#L231

Description: These safeTransfers and events may be emitted unwantonly and multiple times following an initial invocation where a `request.usdcWithdrawable` amount is non-zero. This could be inefficient for legitimate users in terms of gas costs, and a potential griefing vector by malicious users, where they spam multiple valid but useless events from the `SeniorPool` and `Fidu` contracts. Spamming of these events could cause unnecessary additional load on front-ends or other infrastructure dependent on these events.

In general, a cancellation should be a one-shot operation and success should indicate requiring its subsequent deletion and requiring a new withdrawal request be created.

As it stands, using `cancelWithdrawRequest` and `addToWithdrawalRequest` can be used in tandem to effectively modify current request, instead of the former being just oneshot.

Recommendation: The easiest solution would be to require `request.usdcWithdrawable` to be 0 before allowing execution of this function. This would yield multiple invocations not possible and make it a truly one-shot operation. The revert error could advise users to claim any unclaimed amounts before they attempt to cancel any of their remaining fidu from that request, rather then allowing potentially even legitimate users to accidently spam the cancel function but not see any effect where their request is actually deleted under specific conditions.

10.3.2 `SeniorPool` - admin has the ability to set expected future epochs to instead trigger on past times

Context: SeniorPool#L106

Description: This code under specific conditions may allow the admin to set a `headEpoch.endsAt` that is soon to be reached, back to one in the past, equivalent to the old `_epochDuration`.

If for example the current duration were 2 weeks, and about to be reached, and the new duration were to be set to 1 day, the current epoch expected to end shortly, would instead end 13 days ago. This could in turn, yield `_checkpointEpochId` to be higher at the original `headEpoch.endsAt` than expected.

This means that `headEpoch.endsAt` and expected `_checkpointEpochId` cannot be reliably expected upon with any future date and no past date not exceeding the current `_epochDuration`. In addition, deposits correlated with the new epoch that was checkpointed in the past, could've occurred in a future date compared to its end.

In addition, when it comes to the `setEpochDuration` function, it gives a malicious admin griefing potential to also keep extending epoch's indefinitely so they never trigger. There are trust assumptions across these contracts that depend on an honest and trusted operator, when it comes to these admin functions and general configuration needed for these contracts to work correctly.

Recommendation: The fact that no future date cannot be relied upon is a design choice for gas optimization, by allowing for extensions of epochs. However, this lack of finality on past dates for epochs seems to be a potential design flaw. Consider having a precondition here, to only do this specific operation if the new `endsAt` also still at least exceeds or meets the current `block.timestamp`.

10.3.3 `SeniorPool` - various functions have further restrictable visibility with respect to implementation

Context: SeniorPool#L101 SeniorPool#L167 SeniorPool#L284 SeniorPool#L553 SeniorPool#L583 SeniorPool#L601 SeniorPool#L655 SeniorPool#L663 SeniorPool#L673 SeniorPool#L684

Description: A number of functions are declared as public, while never being accessed from an internal source.

Recommendation: It is ideal to restrict functions down to their actual implementation. If a function is declared as public, but only ever accessed externally, it should just be external, which also more clearly communicates its use within the implementation, that it is not re-used internally, which is the expectation with a publicly specified function. In the case of the lines noted under Context, they should be set to external.

10.3.4 `SeniorPool` - Limited "hijacking" of liquidity in the end of an epoch

Context: SeniorPool.sol#L33

Description: While the new proposal prevents the arbitrage opportunity described in the GIP-25 spec, a mitigated version of it is still possible in the case where an upcoming increase in `usdcAvailable` is about to happen right before the epoch changes. It is important to mention that it will be impossible for a front-runner to complete a withdrawal within a single block, this timing opportunity is only possible in cases where liquidity is added right before an epoch is about to change, and that the main impact of such opportunistic behavior is mainly the shortening of the total time the front-runner will have to wait to withdraw usdc from the system, and the potential temporary delay to other users withdrawal requests.

10.3.5 Spec mismatch - Minimum amount request to ignore pro-rata mechanism

Context: SeniorPool.sol#L33

Description: GIP-25 describes the motivation and the spec for the recent change in `SeniorPool`, however, the requirement that below a certain minimum amount the pro-rata mechanism will be ignored is not implemented.

11 December

Conducted December 19th - December 23rd on commit hash b17551fe targeting the following contracts: `CreditLine.sol`, `TranchedPool.sol`, `Schedule.sol` and `MonthlyPeriodMapper.sol`

11.1 Critical Risk

11.1.1 `Schedule._nextPrincipalDueTimeAt` - wrong addition of `gracePrincipalPeriods`

Severity: Critical Risk

Context: Schedule.sol#L226

Description: `_nextPrincipalDueTimeAt` is used to calculate the next principal due time. During this calculation `gracePrincipalPeriods` is being mistakenly added to `principalPeriodAt(startTime, timestamp)`. The error does not affect the example described in the `Schedule` contract mainly due to the combination of using `Math.min`, and `periodsInTerm=12` that ensure `nextPrincipalPeriod` will be 1.

To better understand the issue let's consider the scenario where: `periodMapper = monthly periods` `periodsInTerm = 18` `periodsPerPrincipalPeriod = 6 (halfly)` `gracePrincipalPeriods = 1`

If `_nextPrincipalDueTimeAt` is being called with `timestamp=DEC22(period=9)` then the function will return `startOf(period=18)` instead of `startOf(period=12)`.

Recommendation: The addition of `gracePrincipalPeriods` in `_nextInterestDueTimeAt` should be removed.

11.2 High Risk

11.2.1 Full test suite is neccesary

Severity: High Risk

Description: The test suite at this stage is not complete. Upgradeable contracts and complex systems in general require rigorous testing that includes "happy paths", edge cases, and failure scenarios. Especially this helps with safer future development and upgrading of modules.

As we've seen in some smart contract incidents, a complete test suite can prevent issues that might be hard to find with manual reviews. In addition, some of the issues found by the auditing team could be caught by a full coverage test suite.

11.2.2 `TranchedPool._pay` - `interestAccured` will always be 0, causing an accounting issue in `TranchingLogic`

Severity: High Risk

Context: TranchedPool.sol#L456

Description: Due to an accounting issue in `_pay`, the value of `interestAccured` will always be 0, causing the distribution of zero interest to slices which will cause a cascading issue in the `TranchingLogic` contract which is out of scope, thus it is hard to assess the exact impact of this error given the scope and time frame.

Recommendation: The value that's assigned to `interestAccured` in line 454 should be based on the last checkpoint, and not on the current timestamp.

11.3 Medium Risk

11.3.1 `CreditLine._isLate` - grace periods are incorrectly applied leading to invalid late signals

Severity: Medium Risk

Context: CreditLine.sol#L364

Description: The grace period appears erroneously added to the specified comparison. Adding the grace period to the previous due time, would cause full payments that preced the previous due time, to in fact be considered late, rather than early and appropriately applied, and instead only a late payment exceeding the grace period, is what would yield a non-late state with this logic.

Recommendation: The logic needs to be fixed such that the grace period instead does not consider early payments, late, and also doesn't consider prior paid dues and currently expected dues within grace as being late.

11.3.2 `CreditLine.drawdown` - `lastFullPaymentTime` may be reset to period in past, locking drawdowns

Severity: Medium Risk

Context: CreditLine.sol#L163 CreditLine.sol#L170 CreditLine.sol#L364

Description: If a full payment is done during the lifecycle, and then a drawdown in a future period is done, this will reset the lastFullPaymentTime, to an earlier time. The most impactful effect is that it would make any future drawdowns impossible, essentially locking the drawdown for credit lines where full payments may have been submitted, due to this ability and the combination the `_isLate` gets called at the end to ensure there's no outstanding payments, however, since the time would be set to the beginning term time, it would erase any successful prior payments applied, and therefore incorrectly return that it is late, and cause drawdowns to fail.

Recommendation: Have a guard in place so that `lastFullPaymentTime` cannot be reset to earlier times, and also in this case ensure it is only set to `termStartTime()` for initialization purposes. One possibility could be simply checking that `lastFullPaymentTime` is 0 before initializing it on L163.

11.4 Low Risk

11.4.1 `TranchedPool.pay` - incorrect `amount` variable utilized leading to excess funding pays failing

Severity: Low Risk

Context: TranchedPool.sol#L325 TranchedPool.sol#L328

Description: The spec under `ITranchedPool` for the pay function indicates that it can handle excess payments, and in such cases would just return the excess amounts (ideally it would only ever pull the needed amounts, and that is in fact what this function would do with the fix). However, if excess payments are sent with the current implementation, the function will throw an invalid opcode with its failing assert, that will consume all gas, needlessly punishing users when a successful callpath is possible.

Recommendation: Switch the `amount` variables on the stated lines to instead utilize the `amountToPay` variable, which instead will only use what is required for the subsequent payment allocations, and therefore should never hit the payment remaining assert conditional. This will successfully match the spec indicating that excess payments will work, but it may be more fruitful to update, that the contract will only pull what is needed for payment, if the `amount` variable exceeds the possible payment amount.

11.4.2 `Schedule.withinPrincipalGracePeriodAt` - Spec and implementation mismatch

Severity: Low Risk

Context: Schedule.sol#L147

Description: Based on the spec in the supporting `Schedule.t.sol` test file, this function should return true also for timestamps preceding the startTime, while based on the current implementation it returns false for these.

Recommendation: Change the specified conditional to read `timestamp < startTime ||` instead or appropriately update the spec.

11.5 Gas Optimization

11.5.1 `CreditLine._isLate` - conditional always true for internal variant call

Severity: Gas Optimization

Context: CreditLine.sol#L363

Description: There is a non-zero balance check, however, it will always yield true, on the only current internal callpath which is via the drawdown function on L170, while there is a prior non-zero balance requirement on L157.

Recommendation: Consider removing this check for the internal variant, and instead include the check on the external variant prior to the internal call, as that is the only current case where it may be necessary.

11.5.2 `Schedule` - `SafeCast` is imported but not used

Severity: Gas Optimization

Context: Schedule.sol#L8

Description: The `SafeCast` library is imported but not used which causes the total size of the contract to increase, thus making the contract deployment more expensive gas-wise.

Recommendation: Consider removing unused libraries.

11.5.3 `TranchedPool.distributeToSlicesAndAllocateBackerRewards` - Gas optimization for `totalDeployed`

Severity: Gas Optimization

Context: TranchedPool.sol#L495

Description: The `totalDeployed` state variable is being updated multiple times within the loop inside `distributeToSlicesAndAllocateBackerRewards`, leading to frequent read-write operations on this variable. Instead, we can cache the updated value and only write it to the variable once after the loop has completed.

11.6 Informational

11.6.1 Typos in `CreditLine.sol`

Severity: Informational

Context: CreditLine.sol#L232 CreditLine.sol#L339 CreditLine.sol#L343

Description: Typo in the comment of `totalInterestOwedAt()` and the function name `_lateFeesAccuredOverPeriod()`

Recommendation: Change "beocmes" to "becomes" and `_lateFeesAccuredOverPeriod` to `_lateFeesAccruedOverPeriod`