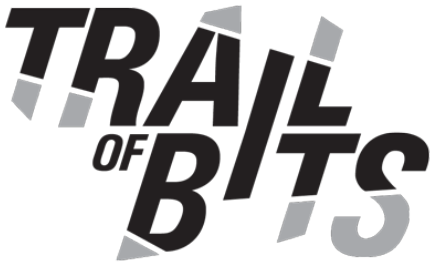

CommunityRewards and UniqueIdentity

Security Assessment

November 10, 2021

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

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

Simone Monica | *Trail of Bits*
simone.monica@trailofbits.com

Changelog:
October 12, 2021: Initial report delivered
November 4, 2021: Fix Log (Appendix D) added

Executive Summary

Executive Summary

From October 4 to October 8, 2021, Goldfinch engaged Trail of Bits to review the security of the Goldfinch V2 protocol. Trail of Bits conducted this assessment over two person-weeks, with two engineers working from commit hash `f79a242` from the `goldfinch-eng/goldfinch-protocol` repository.

The engagement focused on the protocol's new contracts, `CommunityRewards` and `UniqueIdentity`. These contracts introduce an airdropping mechanism and a different approach to user whitelisting, respectively.

Our review resulted in eight findings ranging from medium to informational severity. The most significant issue is related to the roles in the `CommunityRewards` contract. Specifically, the roles are not assigned an admin when the contract is deployed. As a result, if one of these roles were compromised, it could not be reassigned. Three other findings concern the signature hashing scheme used in `UniqueIdentity`. The most significant of these could allow a user to retain a signature retrieved from an off-chain node and to later use it to mint an identity NFT. By delaying the minting operation, a malicious user could prevent the revocation of his or her identity NFT.

In addition to the security findings, we identified code quality issues not related to any particular vulnerability, which are discussed in Appendix C.

Overall, the contracts follow smart contract development best practices and avoid the most common Solidity pitfalls. Additionally, most of the functions are small and have a single purpose. However, the documentation on the system's access controls, the abilities of privileged accounts, and the `CommunityRewards` and `UniqueIdentity` contracts could be improved. Other areas of improvement include the system's tests; several "to-dos" have been left in the unit tests, and the system would benefit from fuzzing of its invariants.

Trail of Bits recommends that Goldfinch take the following steps:

- Address all issues detailed in this report.
- Add Slither to the continuous integration pipeline.
- Identify the important system invariants and test them using Echidna.
- Write high-level documentation on the `UniqueIdentity` and `CommunityRewards` contracts and update the architecture markdown and diagram to include the new contracts.
- Develop documentation on the access controls, the assignment of accounts, and the use cases of the privileged roles.
- Add unit tests for the upgradeability process and complete the tests labeled as to-dos.

Update: On November 4, 2021, Trail of Bits reviewed fixes implemented for the issues in this report. See the results of this fix review in the Fix Log (Appendix D).

Project Dashboard

Application Summary

Name	Goldfinch V2 `UniqueIdentity` and `CommunityRewards` contracts
Version	`f79a242`
Type	Solidity
Platform	Ethereum

Engagement Summary

Dates	October 4–October 8, 2021
Method	Full knowledge
Consultants Engaged	2
Level of Effort	2 person-weeks

Vulnerability Summary

Total High-Severity Issues	0	
Total Medium-Severity Issues	2	■■
Total Low-Severity Issues	2	■■
Total Informational-Severity Issues	4	■■■■
Total	8	

Category Breakdown

Access Controls	1	■
Authentication	3	■■■
Configuration	1	■
Data Validation	1	■
Undefined Behavior	2	■■
Total	8	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Moderate.** The system would benefit from additional high-level documentation on its privileged roles, including the purposes of those roles. We also found that the roles in the `UniqueIdentity` contract are not assigned an admin (TOB-GOL-001).
Arithmetic	**Satisfactory.** The codebase uses `SafeMath`, in line with arithmetic best practices. However, the arithmetic operations would benefit from more thorough testing and verification through fuzzing or symbolic execution.
Assembly Use/Low-Level Calls	**Not applicable.** No assembly was reviewed.
Decentralization	**Moderate.** The privileged roles in the `UniqueIdentity` contract are assigned to the same account upon the contract's deployment. There is no documentation indicating the (type of) account that each role will be assigned to after deployment.
Code Stability	**Moderate.** The code was updated once during the audit and is still undergoing changes.
Contract Upgradeability	**Moderate.** With the exception of the `MerkleDistributor` contract, all of the contracts under review are upgradeable. However, the lack of tests for and documentation on the upgradeability process increases the risk of errors in the process. The system would also benefit from the addition of `slither-check-upgradeability` to the continuous integration pipeline.
Function Composition	**Satisfactory.** The contracts' functions are small, and each serves a single purpose. The `CommunityRewards` contract contains function-level comments. However, the `UniqueIdentity` contract lacks such comments, which would help explain the use of signatures.
Front-Running	**Moderate.** The system's front-running attack vectors should be further analyzed and documented.
Key Management	**Not considered.** We did not review the protocol's key management system.

Monitoring	**Satisfactory.** All functions in the contracts under review emit events. However, there is no incident response plan; nor are there any off-chain components used specifically for behavior monitoring. Such an off-chain component could help the Goldfinch team detect transfers of identity NFTs, which are prohibited (TOB-GOL-008), and identify users whose identity NFTs should be revoked (TOB-GOL-005).
Specification	**Weak.** The contracts lack a high-level specification, and NatSpec comments are not included consistently throughout the codebase.
Testing & Verification	**Moderate.** The contracts have unit and integration tests executed through a continuous integration pipeline. However, several tests are marked as “to-dos” and have not yet been implemented. Additionally, the codebase lacks automated testing such as fuzzing and formal verification.

Engagement Goals

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

Specifically, we sought to answer the following questions:

- Does the `UniqueIdentity` contract adhere to the ERC1155 standard?
- Could a user bypass the restriction on transfers of identity NFTs?
- Is the system vulnerable to reentrancy attacks?
- Is the signature hashing scheme sufficiently robust?
- Are vested rewards calculated correctly?
- Are the access controls implemented correctly and without gaps?
- Are the privileged roles properly managed?
- Are the `UniqueIdentity` contract's `transfer` functions paused when the contract is paused?
- Is there appropriate data validation in place?
- Are there any front-running attack vectors?
- Is it possible to acquire an identity NFT if one's know-your-customer (KYC) verification has been revoked?

Coverage

`CommunityRewards`. The `CommunityRewards` contract holds reward tokens (`GFI` tokens) during their vesting period and mints NFTs to the addresses eligible for rewards.

`MerkleDistributor`. The `MerkleDistributor` contract implements a merkle proof verification scheme used to distribute tokens to users.

`CommunityRewardsVesting`. The `CommunityRewardsVesting` library contains the system's reward-vesting logic.

`UniqueIdentity`. The `UniqueIdentity` contract is an ERC1155-compliant contract that stores the identity verification statuses of addresses. After a user has successfully completed the KYC process, the user can obtain a signature from an off-chain component and use it to mint their ERC1155 identity token.

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

❑ **Set the `OWNER_ROLE` as the admin of the `OWNER_ROLE`, `SIGNER_ROLE`, and `PAUSER_ROLE`.** That way, it will be possible to reassign those roles if (for example) they are compromised. TOB-GOL-001

❑ **Add zero-value checks for all function arguments in `MerkleDistributor.constructor` to ensure that users cannot accidentally set incorrect values, misconfiguring the system.** TOB-GOL-002

❑ **Measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.** TOB-GOL-003

❑ **Use `msg.sender` instead of the `to` parameter in `UniqueIdentity.mint`.** This will prevent front-running, which could cause user confusion. TOB-GOL-004

❑ **Add a timeout value to the signed data so that it will become invalid after a reasonable amount of time (e.g., 10 minutes).** This value will prevent users from using a signature long after receiving it, which could enable users to evade revocation of identity NFTs. TOB-GOL-005

❑ **Include the `UniqueIdentity` contract's address in the signed data of its `mint` and `burn` functions.** This will prevent the reuse of a signature in different instances of the same contract. TOB-GOL-006

❑ **Include a function selector in the signed data of the `UniqueIdentity` contract's `mint` and `burn` functions.** This will prevent the reuse of a signature in different functions. TOB-GOL-007

❑ **Develop strategies for detecting the transfer of an identity NFT through wrapping, and implement a process for taking appropriate action if such a transfer occurs.** This will make it easier to detect and prevent transfers of identity NFTs. TOB-GOL-008

Long Term

❑ **Document the hierarchy of roles and each role's functionality, and create a process for ensuring that the documentation remains up to date.** TOB-GOL-001

❑ **Use Slither, which will catch functions that do not have zero-value checks.** TOB-GOL-002

❑ **Monitor the development and adoption of Solidity compiler optimizations to assess their maturity.** TOB-GOL-003

❑ **Ensure that the system is designed to minimize the risk of front-running.** TOB-GOL-004

❑ **Always consider timing when designing a signature-based process.** TOB-GOL-005

❑ **Design a signature hashing scheme that will protect the system from cross-chain, cross-contract, and cross-function replay attacks.** TOB-GOL-006 TOB-GOL-007

❑ **When implementing restrictions on token transactions, be mindful of the fact that a token can be wrapped in a contract.** TOB-GOL-008

Findings Summary

#	Title	Type	Severity
1	Use of assigned roles that lack admins	Configuration	Medium
2	Lack of zero-value checks in constructor	Data Validation	Low
3	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
4	Risk of mint function front-running and user confusion	Access Controls	Low
5	Lack of a time limit on the use of signatures	Authentication	Medium
6	Lack of contract address in signed data enables signature reuse	Authentication	Informational
7	Lack of function selector in signed data enables signature reuse	Authentication	Informational
8	Nontransferable identity NFT can be wrapped and transferred	Undefined Behavior	Informational

1. Use of assigned roles that lack admins

Severity: Medium | Difficulty: High
Type: Configuration | Finding ID: TOB-GOL-001
Target: `UniqueIdentity.sol, ERC1155PresetPauserUpgradeable.sol`

Description
The contracts use OpenZeppelin's `AccessControl` contract to implement hierarchical role-based access controls. They use three defined roles: `PAUSER_ROLE`, `OWNER_ROLE`, and `SIGNER_ROLE`. The `_setRoleAdmin` function should be used to assign an admin role to each of the three roles. However, the function is not used, and the admin of each role is the `DEFAULT_ADMIN_ROLE`. Furthermore, since that default role has not been assigned to an account, the roles do not actually have an admin and therefore cannot be reassigned.

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

  __ERC1155PresetPauser_init(owner, uri);
  __UniqueIdentity_init(owner);
}

function __UniqueIdentity_init(address owner) internal initializer {
  __UniqueIdentity_init_unchained(owner);
}

function __UniqueIdentity_init_unchained(address owner) internal initializer {
  _setupRole(SIGNER_ROLE, owner);
}
```

Figure 1.1: contracts/protocol/core/UniqueIdentity.sol#L37-L50

The `UniqueIdentity` contract inherits from `ERC1155PresetPauserUpgradeable`. The `UniqueIdentity` contract defines the `SIGNER_ROLE`, and the `ERC1155PresetPauserUpgradeable` contract defines the `OWNER_ROLE` and `PAUSER_ROLE`.

```
function __ERC1155PresetPauser_init_unchained(address owner) internal initializer {
  _setupRole(OWNER_ROLE, owner);
  _setupRole(PAUSER_ROLE, owner);
}
```

Figure 1.2: contracts/external/ERC1155PresetPauserUpgradeable.sol#L47-L50

In other Goldfinch contracts, the owner role is set as the admin of all roles (including the owner role itself).

Exploit Scenario
The signer role is compromised, so the Goldfinch team wants to reassign the role to another account. However, because the default admin role (the signer role's admin) has not been assigned to an account, the team cannot make that change.

Recommendations

Short term, set the owner role as the admin of the owner, signer, and pauser roles.

Long term, document the hierarchy of roles and each role's functionality, and create a process for ensuring that the documentation remains up to date.

2. Lack of zero-value checks in constructor

Severity: Low
Type: Data Validation
Target: `MerkleDistributor.sol`

Difficulty: High
Finding ID: TOB-GOL-002

Description
The `MerkleDistributor`'s constructor fails to validate incoming arguments, so callers can accidentally set important state variables to the zero address.

```
constructor(address communityRewards_, bytes32 merkleRoot_) public {
  communityRewards = communityRewards_;
  merkleRoot = merkleRoot_;
}
```

Figure 2.1: `contracts/rewards/MerkleDistributor.sol#L18-L21`

Once these addresses have been set to the zero address, they cannot be changed. Changing these state variables would require the deployment of a new contract.

Exploit Scenario
Alice, a member of the Goldfinch team, deploys the `MerkleDistributor` contract and sets the `communityRewards` address to `address(0)`. As a result, the `MerkleDistributor` contract needs to be redeployed with the correct `communityRewards` address.

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.

3. Solidity compiler optimizations can be problematic

Severity: Informational
Type: Undefined Behavior
Target: `hardhat.config.js`

Difficulty: Low
Finding ID: TOB-GOL-003

Description
The Goldfinch smart contracts have enabled optional compiler optimizations in Solidity.

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

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

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

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

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

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

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

4. Risk of mint function front-running and user confusion

Severity: Low
Type: Access Controls
Target: `UniqueIdentity.sol`

Difficulty: High
Finding ID: TOB-GOL-004

Description
The `mint` function, which users call to mint an identity NFT, takes a `to` argument that can be set to the address of any account. An attacker could monitor the chain and front-run a user's call to `mint`. The user's call to `mint` would then fail, but he or she would still receive an identity NFT, which could cause confusion.

```
function mint(
  address to,
  uint256 id,
  bytes calldata signature
)
  public
  payable
  override
  onlySigner(keccak256(abi.encodePacked(to, id, nonces[to], block.chainid)), signature)
  incrementNonce(to)
{
  require(msg.value >= MINT_COST_PER_TOKEN, "Token mint requires 0.00083 ETH");
  require(to != address(0), "Cannot mint to the zero address");
  require(id == ID_VERSION_0, "Token id not supported");
  require(balanceOf(to, id) == 0, "Balance before mint must be 0");

  _mint(to, id, 1, "");
}
```

Figure 4.1: `contracts/protocol/core/UniqueIdentity.sol#L52-L69`

There is little incentive for an attacker to front-run a `mint` call, beyond causing confusion. Additionally, executing a call to `mint` costs ETH 0.00083, plus the transaction's gas fee, which further discourages front-running. Still, front-running is possible, though there is a way to prevent it: using `msg.sender` instead of the `to` parameter.

Exploit Scenario
After retrieving the necessary signature, Alice, a user, calls `mint` to obtain an identity NFT. Eve, an attacker monitoring the chain, notices Alice's transaction and front-runs it. Alice's transaction reverts, but she still receives an identity NFT.

Recommendations
Short term, use `msg.sender` instead of the `to` parameter.

Long term, ensure that the system is designed to minimize the risk of front-running.

5. Lack of a time limit on the use of signatures

Severity: Medium
Type: Authentication
Target: `UniqueIdentity.sol`

Difficulty: High
Finding ID: TOB-GOL-005

Description
The signed data passed to the `UniqueIdentity` contract's `mint` function lacks a timeout value and can therefore be used at any time.

```
onlySigner(keccak256(abi.encodePacked(to, id, nonces[to], block.chainid)), signature)
```

Figure 5.1: `contracts/protocol/core/UniqueIdentity.sol#L60`

After completing the KYC process, a user can retrieve a signature from a Goldfinch-controlled off-chain node and use it to call `mint` at any time. If the Goldfinch team decided to revoke a user's identity, it could check whether the user's account had an identity NFT. If the user had not yet called `mint`, the check would return `false`. The team might then assume that the user's KYC verification had been revoked. However, the user's signature would still be valid, so the user could call `mint` to obtain an identity NFT at any time.

As a workaround, a Goldfinch off-chain system could call `mint` for the user and then immediately call `burn`. Assuming these calls were the first calls made for or by the user to the `UniqueIdentity` contract, his or her most recently used nonce would be updated to 2. As a result, the user's original signature (that used to sign data with nonce 1) would no longer be valid.

Another option would be to monitor the events emitted by the `UniqueIdentity` contract and to implement a means of automatically calling `burn` upon the user's call to `mint`. However, that would not be a simple solution.

The Goldfinch team could prevent this issue by adding a timeout value to the signed data. At the end of the timeout period, the signature would become invalid, and the user would have to request a new signature.

Exploit Scenario
Eve completes the KYC process and receives the signature needed for a call to `mint`. Alice, an operator of the Goldfinch system, decides that Eve's KYC verification needs to be revoked and calls `burn` with Eve's address. However, because Eve has not called `mint`, the transaction reverts. Eve can then call `mint` and obtain an identity NFT.

Recommendations
Short term, add a timeout value to the signed data so that it will become invalid after a reasonable amount of time (e.g., 10 minutes).

Long term, always consider timing when designing a signature-based process.

6. Lack of contract address in signed data enables signature reuse

Severity: Informational
Type: Authentication
Target: `UniqueIdentity.sol`

Difficulty: High
Finding ID: TOB-GOL-006

Description
Execution of the `mint` and `burn` functions requires a signature. However, because the address of the `UniqueIdentity` contract is not included in the signed data, the signature could be reused in other instances of the contract.

```
onlySigner(keccak256(abi.encodePacked(to, id, nonces[to], block.chainid)), signature)
```

Figure 6.1: `contracts/protocol/core/UniqueIdentity.sol#L60`

If the Goldfinch team decided to deploy a new `UniqueIdentity` contract, users could reuse a signature from a previous `mint` transaction to execute another call to `mint` in the new version of the contract. In this way, a user whose identity NFT had been burned in the old contract could obtain another in the new contract by calling `mint` without subsequently calling `burn`.

Exploit Scenario
After retrieving the necessary signature, Alice, a user, calls `mint` to obtain an identity NFT. Bob, an operator of the Goldfinch system, deploys a new `UniqueIdentity` contract. Alice calls the new contract's `mint` function, providing the signature that she used in her previous call to `mint`. Alice then receives an identity NFT in the new contract.

Recommendations
Short term, include the contract's address in the signed data.

Long term, design a signature hashing scheme that will protect the system from cross-chain, cross-contract, and cross-function replay attacks.

7. Lack of function selector in signed data enables signature reuse

Severity: Informational
Type: Authentication
Target: `UniqueIdentity.sol`

Difficulty: High
Finding ID: TOB-GOL-007

Description
The data that needs to be signed to call the `mint` and `burn` functions lacks a function selector. This is not currently a problem, as the `mint` and `burn` functions are the only ones in the `UniqueIdentity` contract that use this signature. However, if the contract were upgraded to include another function that required the same data to be signed, the signature could be reused to call that function too.

```
onlySigner(keccak256(abi.encodePacked(to, id, nonces[to], block.chainid)), signature)
```

Figure 7.1: `contracts/protocol/core/UniqueIdentity.sol#L60`

Exploit Scenario
Alice, an operator of the Goldfinch system, upgrades the `UniqueIdentity` contract, adding another function that requires the same data to be signed as the `mint` and `burn` functions. Eve, who has already minted an identity NFT, monitors the chain for Alice's call to `burn` on her account. Eve front-runs the transaction and reuses the old signature to call the new function. Her call to the new function succeeds, and Alice's call to `burn` fails.

Recommendations
Short term, include a function selector in the signed data.

Long term, design a signature hashing scheme that will protect the system from cross-chain, cross-contract, and cross-function replay attacks.

8. Nontransferable identity NFT can be wrapped and transferred

Severity: Informational
Type: Undefined Behavior
Target: `UniqueIdentity.sol`

Difficulty: High
Finding ID: TOB-GOL-008

Description
The ERC1155 standard disallows transfers of identity NFTs. However, it is possible for the recipient of an NFT to be a contract and for a contract to be transferred to another account. As such, it is possible to circumvent this restriction on transfers.

```
function _beforeTokenTransfer(
  address operator,
  address from,
  address to,
  uint256[] memory ids,
  uint256[] memory amounts,
  bytes memory data
) internal override(ERC1155PresetPauserUpgradeable) {
  require(
    (from == address(0) && to != address(0)) || (from != address(0) && to == address(0)),
    "Only mint or burn transfers are allowed"
  );
  super._beforeTokenTransfer(operator, from, to, ids, amounts, data);
}
```

Figure 8.1: `contracts/protocol/core/UniqueIdentity.sol#L87-L100`

Since Goldfinch performs a KYC screening of every account, it can identify which user has executed a transaction. This means that it could identify a user who had circumvented the transfer restriction and take appropriate action (i.e., burn the user's NFT). However, to identify accounts engaging in illicit behavior, Goldfinch would need to actively monitor the chain, which would be nontrivial.

Exploit Scenario
Eve deploys a contract that wraps her identity NFT. Eve then begins the KYC process, providing the contract's address as her address. Goldfinch approves Eve's request and provides her with the signature for a call to `mint`. Eve calls a function of her contract, which calls `UniqueIdentity.mint`. Eve then transfers the ownership of her contract to Mallory, effectively transferring the identity NFT to her.

Recommendations
Short term, develop strategies for detecting the transfer of an identity NFT through wrapping, and implement a process for taking appropriate action if such a transfer occurs.

Long term, when implementing restrictions on token transactions, be mindful of the fact that a token can be wrapped in a contract.

A. Vulnerability Classifications

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

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

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

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

B. Code Maturity Classifications

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

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

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

C. 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.

`MerkleDistributor.sol`

- **Replace the `acceptGrant` function's `account` parameter with `msg.sender`, and remove the `onlyGrantRecipient` modifier from that function.** Because the modifier ensures that the `account` address is that of the message sender, it would become unnecessary if the `account` parameter were removed and `msg.sender` were used directly. These changes would simplify the code and slightly decrease the gas cost of calls to this function.

`CommunityRewardsVesting.sol`

- **Replace the "less than or equal to" sign (`<=`) in the `totalVestedAt` function with a "double equal sign" (`==`).** Because of the way that these values are assigned in `CommunityRewards._grant`, the `end` value cannot be less than the `start` value.

```
startTime: block.timestamp,
endTime: block.timestamp.add(vestingLength),
```

Figure C.1: `contracts/rewards/CommunityRewards.sol#L147-L148`

D. Fix Log

On November 4, 2021, Trail of Bits reviewed the fixes and mitigations implemented by the Goldfinch team for certain issues identified in this report, working from commit `969445e647dbd50216bdbe25baf3e66d3d41dd9c`. The Goldfinch team fixed five issues and acknowledged the other three without fixing them. We reviewed each of the fixes to ensure that the proposed remediation would be effective. For additional information, please refer to the detailed fix log.

#	Title	Severity	Status
1	Use of assigned roles that lack admins	Medium	Fixed
2	Lack of zero-value checks in constructor	Low	Fixed
3	Solidity compiler optimizations can be problematic	Informational	Not fixed
4	Risk of mint function front-running and user confusion	Low	Fixed
5	Lack of a time limit on the use of signatures	Medium	Fixed
6	Lack of contract address in signed data enables signature reuse	Informational	Fixed
7	Lack of function selector in signed data enables signature reuse	Informational	Not fixed
8	Nontransferable identity NFT can be wrapped and transferred	Informational	Not fixed

Detailed Fix Log

Finding 1: Use of assigned roles that lack admins
Fixed. The owner role has been set as the admin of the signer role in the `UniqueIdentity` contract and as the admin of the owner and pauser roles in `ERC1155PresetPauserUpgradeable`.

Finding 2: Lack of zero-value checks in constructor
Fixed. The constructor now checks the `communityRewards_` and `merkleRoot_` parameters to ensure that they are not set to zero.

Finding 4: Risk of mint function front-running and user confusion
Fixed. The `to` parameter has been replaced with `msg.sender`, eliminating the risk of front-running.

Finding 5: Lack of a time limit on the use of signatures
Fixed. The signed data now includes a timeout parameter and will become invalid when the period specified by that parameter ends.

Finding 6: Lack of contract address in signed data enables signature reuse
Fixed. The address of the `UniqueIdentity` contract has been added to the signed data, preventing the reuse of signatures across different contract instances on the same chain.