Catalog Protocol

Security Assessment (Summary Report)

April 5, 2024

Prepared for:
Catalog Technologies Limited

Prepared by: **Richie Humphrey and Vara Prasad Bandaru**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Richie Humphrey, Consultant
richie.humphrey@trailofbits.com

Vara Prasad Bandaru, Consultant
vara.prasad.bandaru@trailofbits.com

Project Timeline

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

Date	Event
February 21, 2024	Pre-project kickoff call
March 1, 2024	Delivery of report draft
March 4, 2024	Report readout meeting
April 5, 2024	Delivery of summary report with fix review appendix

Project Targets

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

garden-sol repository

Repository	https://github.com/trailofbits/audit-catalogfi-stake
Version	`16eb2a9cfa896595b4b145ea8b843297cfab5157`
Type	Solidity
Platform	EVM-based chain

Executive Summary

Engagement Overview

Catalog Technologies Limited engaged Trail of Bits to review the security of the Catalog Finance protocol, which offers cross-chain, peer-to-peer swaps between BTC and WBTC. These swaps are made through the use of hashed timelock contracts (HTLC) and the support of liquidity providers known as fillers, who take the other side of a trade for a fee. In addition, stakers add to the protocol's stability by backing the fillers, who, in turn, reward the stakers with part of their fee.

A team of two consultants conducted the review from February 26 to March 1, 2024, for a total of two engineer-weeks of effort. With full access to source code and documentation, we performed static and dynamic testing of the Catalog Finance protocol, using automated and manual processes.

Observations and Impact

Our review included on-chain components only, specifically the three smart contracts: `GardenStaker`, HTLC, and `GardenFEEAccount`. We did not review any off-chain components, which were out of scope.

The code is high quality, with added attention to security evident in the design. The challenge of this review was not so much understanding the code itself but understanding the interactions with the larger system.

There are no technical design specifications, which can be helpful when developing and analyzing complex systems. While the complexity within each contract is usually well managed, the complexities resulting from interactions with other parts of the system are inadequately documented and in some cases unclear.

Tests are minimal and consist of basic unit tests. There are no integration or invariant tests. The minimal testing combined with the overall system complexity and lack of documentation may have been contributing factors in findings such as TOB-CATALOG-2, TOB-CATALOG-3, and TOB-CATALOG-7.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Catalog Technologies take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Retroactively create a system specification.** The findings discovered during the assessment show that Catalog Finance could benefit from a more formal specification of its components. Much of the information needed to create a specification is already present in code comments or other documentation, such as the diagrams, presented to us during the review.

  Creating a specification would allow engineers to work on new project features with higher confidence and could lead to the discovery of other security issues. Guidance on the creation of specification documents can be found in appendix C.

- **For future iterations, create a system specification before writing code.** Many of the benefits of writing a system specification are lost when the specification is created retroactively. For maximum impact, begin working on the system's specification before writing any code.

- **Create additional unit and fuzz tests based on the specification**. The new specification will identify properties and invariants that should be tested. Creating tests for these properties will prevent an engineer from inadvertently making changes that later break a system property.

- **Perform an additional security review.** During our time-boxed review, we discovered new issues every day, including several on the last day of the engagement. This could be an indication that there are more issues in the code that were not found due to time constraints.

  Consider performing a follow-up review after the system has been refactored and tests have been added. A follow-up review would help verify whether the system's security maturity has improved and could help reveal outstanding bugs that were not discovered during this assessment.

  Furthermore, one of the challenges of this review was understanding the complex interactions between the in-scope code and the other parts of the protocol. As such, if an additional review is performed, consider either including the other components in scope or else allowing additional time to review the overall system.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description
Slither	Slither is a static analysis framework that can statically verify algebraic relationships between Solidity variables. Because Catalog Finance uses Slither as part of its development process, our static analysis did not reveal anything significant.
Echidna	Echidna is a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation. We used Echidna to conduct a stateful invariant fuzzing campaign on the `GardenStaker` contract. While we tested two system properties, we were unable to test additional properties due to time constraints. The result of this testing is detailed below.

Test Results

`GardenStaker.sol`. This contract is used to manage fillers and delegators.

Property	Tool	Result
The SEED token balance of the staker contract equals the net sum of all stakes.	Echidna	**Passed**
All fillers have the `FILLER` role and no other known user has that role.	Echidna	**Passed**

Codebase Maturity Evaluation

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

Category	Summary	Result
Arithmetic	The codebase does not rely heavily on arithmetic, and we did not find any issues with the small amount of existing arithmetic.	**Satisfactory**
Auditing	Events are emitted for most state variable changes and critical functions. However, we found that the `settle` function does not emit an event. The incident response plan is still in progress, so it was considered out of scope. We included recommendations for creating such a plan in appendix D.	**Satisfactory**
Authentication / Access Controls	While traditional permissioned functions are not used, most functions involve the use of ECDSA signatures or an HTLC secret word.	**Satisfactory**
Complexity Management	The individual contracts are generally simple, and functions have a clear, singular purpose. However, various areas of complexity are not properly highlighted or documented via inline comments within the code or in external documentation. Most important are the nuances and risks from the interactions with off-chain components. Certain concepts were unclear, such as the inconsistent use of the term “expiry” (TOB-CATALOG-6). Additionally, the term “delegate” is used incorrectly throughout the codebase and in the documentation, as described in appendix E.	**Moderate**

Decentralization	We did not identify any significant centralization risks in the in-scope code. However, we reviewed only a part of the larger system, which includes off-chain components that may contain centralization risks.	**Satisfactory**
Documentation	There are NatSpec comments on most external functions. There are also helpful `README`s in the codebase. However, these forms of documentation do not include important information about interactions with the rest of the protocol or other nuances. We recommend creating full user and developer documentation such as a wiki website and detailed technical specifications.	**Moderate**
Low-Level Manipulation	Low-level code is not used in this codebase.	**Not Applicable**
Testing and Verification	There are some basic unit tests that largely center on core functionality and happy paths. The codebase would benefit from additional testing, including invariant tests and full system integration tests. Stateful invariant tests could have identified issues such as TOB-CATALOG-7 and TOB-CATALOG-8.	**Weak**
Transaction Ordering	Much of the processing takes place off-chain and cannot be front run. There is risk from a chain reorganization, but this risk has been considered and mitigations, such as the expiration difference between the initiator HTLC and the redemption HTLC, have been put in place. There is a lack of documentation to address these risks, and we did not find any testing related to transaction ordering.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Filler liquidity vulnerable to a DoS attack	Denial of Service	High
2	Anyone can cause the GardenFeeAccount template to self-destruct	Configuration	High
3	Fees may be withdrawn prior to initiating an order	Configuration	Medium
4	A creator can claim fees without redeeming the order	Configuration	Medium
5	Delegators can extend their stake at a higher voting multiple	Data Validation	Low
6	Inconsistent use of the term “expiry”	Data Validation	Informational
7	Improper handling of delegateStakeIDs disables critical functionality	Data Validation	High
8	A user cannot renew their stake for the maximum duration	Data Validation	Low
9	Delegators lose rewards when HTLCs expire before others in a channel	Data Validation	Medium

Detailed Findings

1. Filler liquidity vulnerable to a DoS attack	
Severity: **High**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-CATALOG-1
Target: `GardenHTLC.sol#L143–L155`	

Description

Because there is no penalty for refunding orders, an attacker can initiate an order with no intention of redeeming it or revealing the secret word. If an attacker created enough orders, all of the filler liquidity would be used and the system would be effectively disabled.

```
function refund(bytes32 orderID) external {
    Order storage order = orders[orderID];

    require(order.redeemer != address(0), "GardenHTLC: order not initiated");
    require(!order.isFulfilled, "GardenHTLC: order fulfilled");
    require(order.initiatedAt + order.expiry < block.number, "GardenHTLC: order not
expired");

    order.isFulfilled = true;

    emit Refunded(orderID);

    token.safeTransfer(order.initiator, order.amount);
}
```

Figure 1.1: The `refund` function being called after the order has expired (`GardenHTLC.sol#L143–L155`)

The `refund` function can be called at any time after the order has expired.

Exploit Scenario

1. Eve initiates a swap from BTC to WBTC, which creates an HTLC on the Ethereum side that only she can redeem with her chosen secret word.
2. Eve repeats this process until there are no more fillers available.
3. Eve does not call `redeem`, does not reveal her secret code, and then calls `refund` on the HTLC once the expiration has passed.

Recommendations

Short term, there is no simple solution to mitigate this issue. However, the following are some mitigation ideas to consider:

- Require the initiator to make a deposit that is returned upon successful redemption.
- Charge a fee for initiating.
- Require initiators to stake a minimum amount of SEED.
- Have any mechanism used to dissuade this attack be triggered by low filler capacity. For example, a fee is charged only when filler capacity is below a certain level.

Long term, create a technical design specification that clearly illustrates the interactions between the protocol's components, the intended behavior, and the associated risks.

2. Anyone can cause the GardenFeeAccount template to self-destruct	
Severity: **High**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-CATALOG-2
Target: `fee/GardenFEEAccountFactory.sol`, `fee/GardenFEEAccount.sol`	

Description

The `GardenFEEAccountFactory` does not initialize the `GardenFEEAccount` template contract, which could allow an attacker to cause the template to self-destruct. As a result, tokens owned by the template's clones would be lost.

The `GardenFEEAccount` contract is used to create payment channels, which are used as counterfactual wallets, and the tokens are deposited before the channel is created. The users can request the `GardenFEEAccountFactory` contract to create the channel, and the `GardenFEEAccountFactory` contract will create a clone from the template contract (figure 2.1).

```
 139    function _create(address funder, address recipient) internal returns
(GardenFEEAccount) {
[...]
 144
 145        address channel = template.cloneDeterministic(salt);
 146        channels[recipient] = GardenFEEAccount(channel);
 147        channels[recipient].__GardenFEEAccount_init(token, funder, recipient,
feeAccountName, feeAccountVersion);
```

Figure 2.1: The `_create` function being used to create a payment channel (`contracts/fee/GardenFEEAccountFactory.sol#139–147`)

The user can call the `GardenFEEAccount.close` or `GardenFEEAccount.settle` functions to close the channel. Both the functions will call the `closeChannel` function, which transfers the tokens and causes the contract to self-destruct (figure 2.2).

```
217    function closeChannel(uint256 amount_) internal {
218       token.safeTransfer(recipient, amount_);
219       token.safeTransfer(funder, totalAmount());
220
221       factory.closed(recipient);
222
223       selfdestruct(payable(recipient));
224    }
```

Figure 2.2: The `closeChannel` function transferring the tokens to the user and causing the `GardenFeeAccount` contract to self-destruct (`contracts/fee/GardenFEEAccount.sol#217–224`)

The template contract is created in the constructor of `GardenFEEAccountFactory` (figure 2.3). The constructor does not initialize the template contract, nor does it disable the initializers.

```
34    constructor(
[...]
39    ) {
40       token = token_;
41       feeManager = feeManager_;
42       template = address(new GardenFEEAccount());
43       feeAccountName = feeAccountName_;
44       feeAccountVersion = feeAccountVersion_;
45    }
```

Figure 2.3: The `constructor` failing to initialize the template contract or disable initializers (`contracts/fee/GardenFEEAccountFactory.sol#34–45`)

As a result, an attacker can initialize the template contract with their addresses and close the channel, destroying the template contract. Because the clones perform `delegatecall` on the template, all the calls to the clones will become no-operations, and the tokens that are deposited into the channels will be lost.

Exploit Scenario

The fee manager has a channel with each filler and each delegator. The value of the tokens deposited in the channels is approximately 1 million dollars. Eve, an attacker, initializes the template contract, with the funder and recipient set to her address. Eve calls `GardenFEEAccount.close` on the template contract with the signed `close` message. The template contract self-destructs. All calls to the clones become no-operations and the tokens in the channels are lost.

Recommendations

Short term, disable the initializers of the template contract using the `_disableInitializers` function in the constructor.

Long term, add unit tests to ensure that all the initializable contracts deployed in a function are initialized in the same transaction.

3. Fees may be withdrawn prior to initiating an order

Severity: **Medium**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-CATALOG-3
Target: Catalog Finance system	

Description

The fee manager distributes the fees to the delegators using HTLCs. The HTLCs have the same `secretHash` as the swap HTLCs, which restricts the delegators to claiming the fee if and only if the swap is completed. The fee payment HTLCs are signed by the fee manager before the creator initiates the order.

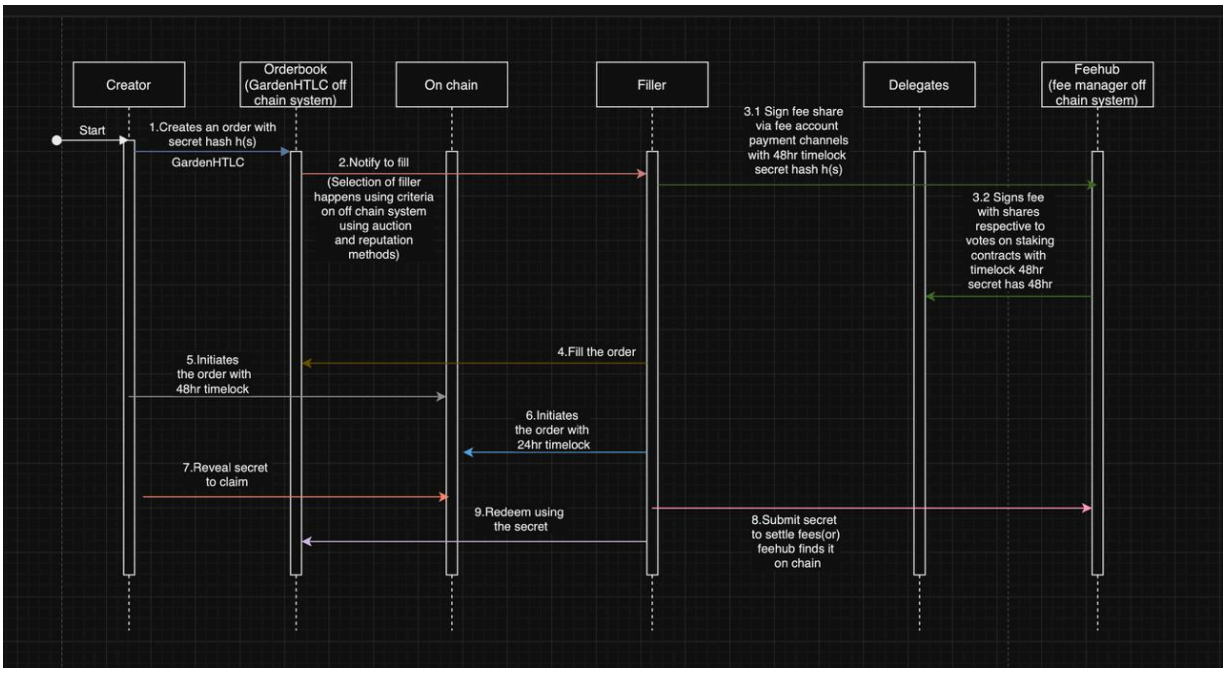

Figure 3.1: System flow diagram provided by the Catalog Finance team showing the fee manager signing the HTLCs before the order is initiated

Because the fee payment HTLCs are preapproved using the `secretHash` provided by the creator, a malicious delegator can create the order off-chain and use the secret word to claim the fees without ever initiating the swap.

Exploit Scenario

Eve, a delegator of the most reputed filler, creates an off-chain order for 10 million dollars worth of BTC. The filler charges a high fee of 10% to complete the transaction and Eve agrees. The fee manager signs a claim message using the `secretHash` provided by Eve.

Eve, without initiating the order on-chain, claims the fee of 10,000 dollars using the signed claim message. Eve repeats the process and steals assets from the filler.

Recommendations

Short term, update the system with the following changes:

- In step 3.2 (figure 3.1), allow the fee manager to share the signed claim message only with the filler.
- After order initiation in step 5 (figure 3.1), let the fee manager share the claim message with the delegators.

Long term, create a complete design specification and consider performing a design review of the system. Doing so will help identify these types of issues.

4. A creator can claim fees without redeeming the order	
Severity: **Medium**	Difficulty: **Medium**
Type: Configuration	Finding ID: TOB-CATALOG-4
Target: Catalog Finance system	

Description

The Catalog Finance team identified this issue simultaneously during the review; the issue is included in the report for completeness.

The timelock for the fee payment HTLCs is set to 48 hours, which is the same as the timelock for the swap HTLCs that lock the creator's tokens (figure 4.1).

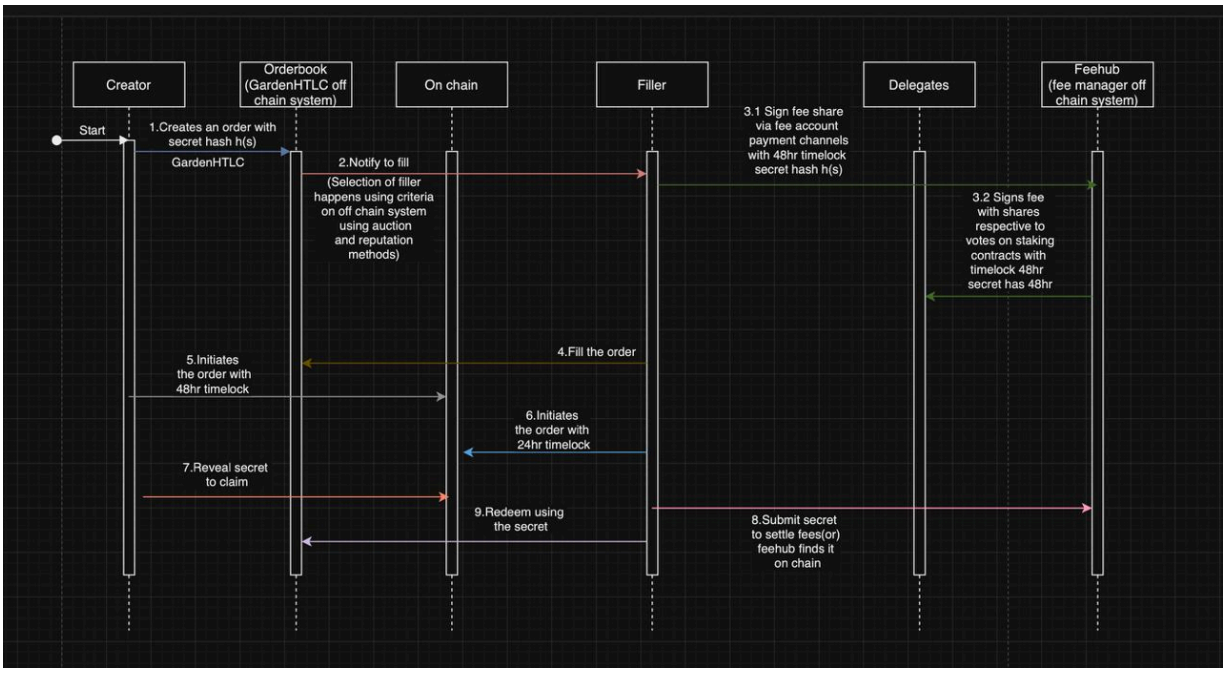

Figure 4.1: System flow diagram provided by the Catalog Finance team showing the timelocks in steps 3.2 and 5

The swap HTLCs and the fee payment HTLCs use the same secret, and the secret is known only to the creator. The creator can refund the tokens locked in the swap HTLCs without revealing the secret and use the secret to claim fees for an incomplete order.

Exploit Scenario

Eve, a delegator of the most reputed filler, creates an off-chain order for 10 million dollars worth of BTC. The filler charges a high fee of 10% to complete the transaction and Eve agrees. The fee manager signs a claim message using the `secretHash` provided by Eve.

Eve creates an order depositing WBTC on Ethereum a few minutes before the fee manager signs the claim message. After 48 hours, Eve calls the `refund` method on the `GardenHTLC` contract and claims the fee using the signed claim message.

Recommendations

Short term, set the timelock for fee payment HTLCs to 45 hours while restricting the maximum delay to 12 hours between steps 3 and 5 and to 6 hours between steps 5 and 6 (figure 4.1).

Long term, create a complete design specification and consider performing a design review for the system. Doing so will help identify these types of issues.

5. Delegators can extend their stake at a higher voting multiple

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CATALOG-5
Target: `DelegateManager.sol#L154-L169`	

Description

When a delegator extends their stake for less time than they had originally staked, they retain the higher voting multiplier, giving them more votes.

As shown in figure 5.1, the logic handles the case where the new multiplier is higher than the current multiplier inferred from the ratio of votes to units.

```
function extend(bytes32 stakeID, uint256 newLockBlocks) external {
    Stake memory stake = stakes[stakeID];

    require(stake.owner == _msgSender(), "DelegateManager: caller is not the owner
of the stake");
    require(stake.expiry > block.number, "DelegateManager: expired stake");

    uint8 multiplier = _calculateVoteMultiplier(newLockBlocks);
    if (multiplier > stake.votes / stake.units) {
        stake.votes = multiplier * stake.units;
    }
    stake.expiry = multiplier == uint8(7) ? MAX_UINT_256 : stake.expiry +
newLockBlocks;

    stakes[stakeID] = stake;

    emit StakeExtended(stakeID, newLockBlocks);
}
```

Figure 5.1: The `extend` function logic retaining the new multiplier (`DelegateManager.sol#L154–L169`)

However, the case where the new multiplier is lower is not handled and the higher multiplier is retained.

Exploit Scenario

1. Eve stakes 100 units for four years and receives a voting power multiplier of 4.

2. Three years, 11 months, and 29 days later, Eve calls `extend` for six more months. She retains the higher voting power multiplier instead of the multiplier of 1 she would usually receive if she staked for six months.

Recommendations

Short term, consider preventing users from extending for a period less than they originally staked.

Long term, create a technical design specification that describes intended behavior and highlights edge cases like this.

6. Inconsistent use of the term “expiry”

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CATALOG-6
Target: `GardenHTLC.sol`	

Description

The term “expiry” is used to mean the expiration block number in the `GardenStaker` contract, but in the `GardenHTLC` contract, it is used to mean a duration, or number of blocks. If someone created an HTLC and used “expiry” in the way it is defined in the code, they would be unable to collect a refund for over 7 years due to the number of blocks that would have to pass before the expiration block was reached.

The NatSpec comments in `GardenHTLC` incorrectly refer to the term “expiry” as a block number throughout the contract. The term is also incorrectly defined in the `README`:

```
### Initiator Functions:
1. `initiate`: Initiators can create an order by providing the necessary order
parameters. The order serves as one-half of the atomic swap commitment.
2. `refund`: Initiators can refund the locked assets after the expiry block number
if the redemption has not occurred.
3. `initiateWithSignature`: delegate can create an order for initiators with `EIP712
signature` signed by Initiators by providing the redeemer's address, expiry block
number, amount of tokens to trade, and the secret hash for redemption.
```

Figure 6.1: NatSpec comments referring to “expiry block number” (`htlc/README.md#L11–14`)

However, ”expiry” actually represents a duration or number of blocks, as confirmed by Catalog Technologies and the check within the `refund` function shown in figure 6.2. The “expiry” is added to the block number the HTLC was initiated at.

```
function refund(bytes32 orderID) external {
    …
    require(order.initiatedAt + order.expiry < block.number, "GardenHTLC: order not
expired");
```

Figure 6.2: The `refund` function showing the correct use of `expiry` (`GardenHTLC.sol#L148`)

Recommendations

Use the correct definition in the documentation, and choose a better variable name that conveys the correct meaning and is different from the variable used in `GardenStaker`.

7. Improper handling of delegateStakeIDs disables critical functionality	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CATALOG-7
Target: `GardenStaker.sol`	
Found by: Slither	

Description

If a filler deregisters and calls `refund`, all associated stakes become permanently broken. These stakes can no longer be used in critical functions such as `refund` or `changeVote`. Furthermore, if the deregistered filler were to re-register, it would not change the situation. The previous stakes would still not be delegated to the filler and could not be refunded or change the vote. They are permanently locked in the contract and do not delegate any voting power to any fillers.

To understand this issue, we need to look at the Open Zeppelin `set` data structure and the inner workings of Solidity. The `Filler` struct shows that the `delegateStakeIDs` struct member uses OpenZeppelin's `EnumerableSet` type.

```
    struct Filler {
        uint16 feeInBips;
        uint256 stake;
        uint256 deregisteredAt;
        EnumerableSet.Bytes32Set delegateStakeIDs;
    }
```

Figure 7.1: The `Filler` struct showing use of the `EnumerableSet` type (`BaseStaker.sol#L26–L31`)

OpenZeppelin's `EnumerableSet` data structure implementation uses the Solidity array and mapping types under the hood.

```
struct Set {
    // Storage of set values
    bytes32[] _values;
    // Position of the value in the `values` array, plus 1 because index 0
    // means a value is not in the set.
    mapping(bytes32 => uint256) _indexes;
}
```

Figure 7.2: Solidity array and mapping types used in `EnumerableSet` (`EnumerableSet.sol#L51–L57`)

The `Filler` struct that uses this set is created when a filler registers and is stored in the `fillers` mapping. When a filler deregisters and then calls `refund`, the entry in the `fillers` mapping is deleted.

```
function refund(address filler_) external {
    Filler storage filler = fillers[filler_];

    require(filler.deregisteredAt != 0, "FillerManager: not registered");
    require(filler.deregisteredAt + FILLER_COOL_DOWN < block.number, "FillerManager:
cooldown not passed");

    delete (fillers[filler_]);
```

Figure 7.3: Entry in `fillers` mapping deleted (`FillerManager.sol#L68–L74`)

However, when the `delete` function is called on a struct containing a mapping, the mapping elements are not deleted, as detailed in the Solidity documentation. For this reason, OpenZeppelin provides the following warning:

> *Trying to delete such a structure from storage will likely result in data corruption, rendering the structure unusable.*

This is a problem because the `remove` function for the set uses the `_indexes` mapping (shown above in figure 7.2) to determine the index of the set element within the `_values` array.

```
function _remove(Set storage set, bytes32 value) private returns (bool) {
    // We read and store the value's index to prevent multiple reads from the same
storage slot
    uint256 valueIndex = set._indexes[value];

    if (valueIndex != 0) {

        {...omitted for brevity...}
        // Delete the slot where the moved value was stored
        set._values.pop();
```

Figure 7.4: The `valueIndex` variable is nonzero. (`EnumerableSet.sol#L83–L106`)

Since the mapping is not deleted, the `valueIndex` variable (figure 7.4) will be a nonzero value, even after the struct containing the set has been deleted. Later in the same `remove` function, the `pop` method is called on the underlying array, but the underlying array length was zeroed out when the struct was deleted. This results in a panic, causing the transaction to revert.

In both `refund` and `changeVote`, `remove` is called on the `delegateIds` member of the `Filler` struct, but, as explained above, this will revert when the filler is deleted.

If the filler re-registers, the new `Filler` struct will no longer contain the set of `delegateStakeIDs`, so the voting power from the stakers will not be acknowledged. Since `changeVote` cannot be called for the reasons explained above, those stakes still do not delegate to any filler.

Exploit Scenario

1. Alice registers to become a filler.
2. Bob stakes 100,000 units for six months and delegates voting power to Alice.
3. Alice decides to deregister, and after the cooldown period, she refunds her stake, which removes her from the system.
4. Bob cannot call `changeVote` on his stake, and six months later, he also cannot call `refund`.
5. Alice decides to re-register as a filler.
6. Bob's situation remains unchanged. His stake is still not delegated to Alice, it cannot be delegated to anyone, and he cannot refund his stake.

Recommendations

Short term, set the `FILLER_COOLDOWN` constant to a reasonable duration to allow the delegators to change their votes. Additionally, update the logic in the `changeVote` and `refund` functions to iterate through the `delegateStakeIDs` in the `Filler` struct and remove each element before deleting the struct.

Long term, create stateful invariant tests. While this issue is complex in nature, it is easily identified with basic fuzz tests. Additionally, run Slither to detect this and other issues.

8. A user cannot renew their stake for the maximum duration

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CATALOG-8
Target: `GardenStaker.sol`	

Description

If a user attempts to renew their expired stake to the maximum duration, the transaction will revert.

```
function renew(bytes32 stakeID, uint256 newLockBlocks) external {
    Stake memory stake = stakes[stakeID];

    require(stake.owner == _msgSender(), "DelegateManager: incorrect owner");
    require(stake.expiry < block.number, "DelegateManager: stake not expired");

    uint8 multiplier = _calculateVoteMultiplier(newLockBlocks);
    stake.expiry = block.number + newLockBlocks;
    stake.votes = multiplier * stake.units;

    stakes[stakeID] = stake;

    emit StakeRenewed(stakeID, newLockBlocks);
}
```

Figure 8.1: The `renew` function reverting due to an arithmetic overflow (`DelegateManager.sol#L130–L143`)

This is because when the user wants to renew for the maximum duration, they must pass the maximum `uint256` value. This will always cause this function to revert due to an arithmetic overflow when the `newLockBlocks` variable is added to the current block number.

Exploit Scenario

Alice's stake expires, and when she tries to renew it for the maximum duration, the transaction reverts.

Recommendations

Short term, when the maximum duration is selected, set `stake.expiry` to `type(uint).max`.

Long term, create tests that call functions with fuzzed parameters. Fuzz testing will identify issues such as this one.

9. Delegators lose rewards when HTLCs expire before others in a channel

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CATALOG-8
Target: `GardenFEEAccount.sol`	

Description

In situations where HTLCs expire before others in a payment channel, the expired HTLCs can no longer be claimed by delegators.

```
function claim(
    uint256 amount_,
    uint256 nonce_,
    HTLC[] memory htlcs,
    bytes[] memory secrets,
    bytes memory funderSig,
    bytes memory recipientSig
) external {
    require(htlcs.length == secrets.length, "GardenFEEAccount: invalid input");
    bytes32 claimID = claimHash(amount_, nonce_, htlcs);

    uint256 localSecretsProvided = 0;
    for (uint256 i = 0; i < htlcs.length; i++) {
        if (htlcs[i].timeLock > block.number && sha256(secrets[i]) ==
htlcs[i].secretHash) {
            localSecretsProvided++;
            amount_ += htlcs[i].sendAmount;
            amount_ -= htlcs[i].recieveAmount;
        }
    }
```

Figure 9.1: Expired elements are not included in the amount sent to the caller (`GardenFEEAccount.sol#L120–L138`)

The `claim` function iterates through the submitted `htlcs` array and checks each element for expiration. If any are expired, they are not included in the final amount sent to the caller.

Exploit Scenario

Time (hr)	Action
0	Alice requests a swap. The `feeManager` includes the fee `HTLC-A` for this swap in the claim message and sets the timelock to `0 + 48 hours`.
6	Alice calls the `initiate` function on the source chain.
12	The `Filler-1` filler selected for the order calls the `initiate` function on the destination chain.
19	Bob requests a swap. The `feeManager` increments the nonce and includes the fee `HTLC-B` for the second swap, setting the timelock to `(19 + 48 = 67 hours)`.
19	The `feeManager` goes offline for `t = 19` and `t = 20`.
20	Alice redeems the tokens and reveals the secret.
20	Charlie, a delegator, calls `GardenFEEAccount.claim` with `A`'s secret and uses the state for `HTLC-A` and `HTLC-B`. He waits for Bob's swap to complete for the secret to be revealed.
39	After 19 hours, Bob's swap is complete and the secret is revealed.
39	Charlie uses B's secret along with A's secret, but he receives the rewards only for Bob's swap and loses the rewards in `HTLC-A` since it has expired.

Recommendations

Short term, there is no simple solution for this issue. One idea is to track expired HTLCs in the channel contract state and account for them during the claim process.

Long term, create stateful invariant tests that call multiple functions at different time intervals using fuzzed parameters.

A. Vulnerability Categories

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

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

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

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

B. Code Maturity Categories

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

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

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

C. Design Specification Guidance

This section provides generally accepted best practices and guidance for how to write design specifications.

A good design specification serves three purposes:

1. **Detect bugs and inconsistencies in a proposed system architecture before any code is written.** Codebases that are written without adequate specifications are often littered with snippets of code that have lost their relevance as the system's design has evolved. Without a specification, it is exceedingly challenging to detect such code snippets and remove them without extensive validation testing.

2. **Reduce bugs in system implementations**. In systems without a specification, engineers must divide their attention between designing the system and implementing it in code. In projects requiring multiple engineers, engineers may make assumptions about how another engineer's component works, creating an opportunity for bugs to be introduced.

3. **Improve the maintainability of system implementations and reduce the likelihood that future code changes will introduce bugs.** Without an adequate specification, new developers need to spend time "on-ramping," where they explore the code and understand how it works. This process is highly error-prone and can lead to incorrect assumptions and the introduction of bugs.

Low-level designs may also be used by test engineers to create property-based fuzz tests or by auditors to reduce the amount of time needed to audit a specific protocol component.

Specification Construction

A good specification must describe system components in enough detail that an engineer unfamiliar with the project can use the specification to implement these components. The level of detail required to achieve this can vary from project to project, but generally, a low-level specification will, at a minimum, include the following details:

- How each system component (e.g., a contract or plugin) interacts with and relies on other components

- The actors and agents that participate in the system; how they interact with the system; and their permissions, roles, authorization mechanisms, and expected known-good flows

- The expected failure conditions the system may encounter and how those failures are mitigated (including failures that are automatically mitigated)

- Specification details for each function the system will implement, including the following:
    - A description of the function's purpose and its intended use
    - A description of the function's inputs and the various validations that are performed against each input
    - Any specific restrictions on the function's inputs that are not validated and not obvious
    - Any interactions between the function and other system components
    - The function's various failure modes, such as failure modes for queries to a Chainlink oracle for a price (e.g., stale price, oracle disabled)
    - Any authentication or authorization required by the function
    - Any function-level assumptions that depend on other components behaving in a specific way

In addition, specifications should use standardized RFC-2119 language as much as possible. This language pushes specification authors towards a writing style that is both detailed and easy to understand. One relevant example is the ERC-4626 specification, which uses RFC-2119 language and provides enough constraints on implementers so that a vault client for a single implementation may be used interchangeably with other implementations.

D. Incident Response Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Catalog Technologies will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific system components.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop "muscle memory." Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

E. Code Quality Recommendations

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

- **Remove the code snippets highlighted in figures E.1 and E.2.** The `BaseStaker` and `FillerManager` contracts do not have variables of type `EnumerableSet.AddressSet`.

```
14    abstract contract BaseStaker is AccessControl {
15        using EnumerableSet for EnumerableSet.AddressSet;
```

Figure E.1: contracts/stake/BaseStaker.sol#14–15

```
14    abstract contract FillerManager is BaseStaker {
15        using EnumerableSet for EnumerableSet.AddressSet;
```

Figure E.2: contracts/stake/FillerManager.sol#14–15

- **Avoid relying on the internals of the `EnumerableSet` library, and update the code in figures E.3 and E.4 to use the equivalent `values` method.** Libraries' internals are susceptible to breaking changes more often than public methods are.

```
69      return (f.feeInBips, f.stake, f.deregisteredAt,
        f.delegateStakeIDs._inner._values);
```

Figure E.3: contracts/stake/BaseStaker.sol#69

```
178     bytes32[] memory delegates =
fillers[filler].delegateStakeIDs._inner._values;
```

Figure E.4: contracts/stake/DelegateManager.sol#178

- **Use block timestamps to measure time passed instead of using the block numbers in the system.** Using block timestamps removes the need for block time approximation and will be inline with general development best practices.

- **Correct the error message in figure E.5.** The condition checks whether the filler has deregistered, but the error message incorrectly implies that the condition checks whether the filler has registered.

```
71      require(filler.deregisteredAt != 0, "FillerManager: not
        registered");
```

Figure E.5: contracts/stake/FillerManager.sol#71

- **Add documentation for mitigations of vulnerabilities that are implemented in one component but applied to a different component.** The `GardenFeeAccount.close` function is vulnerable to a signature replay attack when considered individually; however, the use of a nonce in the `GardenFeeAccountFactory` contract to deploy the clones mitigates this vulnerability in the system. Including information about the mitigation in the `close` function would allow the system to be easily audited and also prevent new developers from unknowingly removing these mitigations. The documentation should also be added for other similar mitigations in the system.

- **Add documentation to the `GardenHTLC.redeem` function explaining why it allows the user to redeem the order even after it has expired.** Documentation improves understanding of the codebase and also prevents new developers from introducing unintended checks into the system.

- **Add zero-address checks to the following contract constructors and initializers**. These checks help identify incorrect deployments of the contracts where some of the address fields are accidentally set to the zero address.

```
72    function __GardenFEEAccount_init_unchained(
73       IERC20Upgradeable token_,
74       address funder_,
75       address recipient_
76    ) internal onlyInitializing {
77       token = token_;
78       funder = funder_;
79       recipient = recipient_;
80       factory = IGardenFEEAccountFactory(msg.sender);
81    }
```

Figure E.6: contracts/fee/GardenFEEAccount.sol#72–81

```
34    constructor(
35       IERC20Upgradeable token_,
36       address feeManager_,
37       string memory feeAccountName_,
38       string memory feeAccountVersion_
39    ) {
40       token = token_;
41       feeManager = feeManager_;
```

Figure E.7: contracts/fee/GardenFEEAccountFactory.sol#34–41

```
 46    constructor(address seed, uint256 delegateStake, uint256 fillerStake,
uint256 fillerCooldown) {
 47       SEED = IERC20(seed);
```

Figure E.8: contracts/stake/BaseStaker.sol#46–47

```
 32    constructor(string memory name, string memory symbol, address
gardenStaker_) ERC721(name, symbol) {
 33        gardenStaker = IGardenStaker(gardenStaker_);
```

Figure E.9: contracts/Flower.sol#32–33

- **Add missing NatSpec comments for the following:**
    - The `GardenFEEAccount.__GardenFEEAccount_init` function
    - The `GardenFEEAccount.__GardenFEEAccount_init_unchained` function
    - The return value of the `GardenFEEAccount.claimHash` function
    - The `GardenFEEAccountFactory.claimed` function
- **Update the `DelegateManager.extend` function to revert if the current lock-in period for the stake is the `MAX_UINT_256` constant**. Reverting will be inline with the expected behavior because `MAX_UINT_256` represents an infinite lock-in period and cannot be extended. Additionally, the function emits the `StakeExtended` event without any state updates when the new lock-in period is finite, which could cause the off-chain system to consider the incorrect data in the event.
- **Correct the use of "delegate" in the codebase and documentation**. The term "delegate" is incorrectly used to refer to users who stake their tokens in the system. Using the term "delegator" or "staker" will eliminate the confusion.
- **Update the `DelegateManager.getVotes` function to check that the filler is not deregistered**. Deregistered fillers will still have `stakeIds` in the `delegateStakeIDs`, which should not be included in this calculation.
- **Update the `FillerManager.register` function to take `feeInBips` as a parameter.** This will be an added convenience to the filler and eliminate the need for the filler to create another transaction to update the fee.

F. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From March 15 to March 18, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Catalog Finance team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

As mentioned in the Executive Summary, this review was unique in that it primarily involved understanding how the smart contracts interacted with the off-chain components, which were out of scope for this review. Several of the issues in our report (TOB-CATALOG-1, TOB-CATALOG-3, and TOB-CATALOG-4) were identified as risks to the overall protocol due to how they are used. For these, the mitigation involved changes to the behavior of out-of-scope, off-chain components. As such, further investigation would be needed to verify these updates.

In summary, of the nine issues described in this report, Catalog Finance has resolved six issues. The fixes for the remaining three issues are out of scope for this review and require further investigation to confirm their resolution. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Filler liquidity vulnerable to a DoS attack	Further Investigation Required
2	Anyone can cause the GardenFeeAccount template to self-destruct	Resolved
3	Fees may be withdrawn prior to initiating an order	Further Investigation Required
4	A creator can claim fees without redeeming the order	Further Investigation Required
5	Delegators can extend their stake at a higher voting multiple	Resolved
6	Inconsistent use of the term “expiry”	Resolved

7	Improper handling of delegateStakeIDs disables critical functionality	**Resolved**
8	A user cannot renew their stake for the maximum duration	**Resolved**
9	Delegators lose rewards when HTLCs expire before others in a channel	**Resolved**

Detailed Fix Review Results

TOB-CATALOG-1: Filler liquidity vulnerable to a DoS attack
Further investigation required. The proposed fixes are outside the scope of this review and require further investigation to confirm resolution.

The Catalog Finance team provided the following context for this finding's fix status:

> *In beta (i.e., invite-only release of our platform), we've a rate limit of 1.5 BTC/WBTC per address in 24 hours. Further down the line when we release a public version of Catalog Finance, our fee collection system (off-chain) will mitigate this issue by charging a certain nonrefundable platform fee per swap created by any user. Also, our HTLCs' timelocks are defined to help minimize the impact of such an attack, as the adversary must lock their funds for two times the fillers' timelock.*

TOB-CATALOG-2: Anyone can cause the GardenFeeAccount template to self-destruct
Resolved in PR #2. The implementation contract now includes an `initialize` function that calls `disableInitializers`, which prevents future attempts to initialize the contract. The new `initialize` function is now called from the constructor when `GardenFEEAccountFactory` is deployed.

TOB-CATALOG-3: Fees may be withdrawn prior to initiating an order
Further investigation required. The proposed fixes are outside the scope of this review and require further investigation to confirm resolution.

The Catalog Finance team provided the following context for this finding's fix status:

> *Took a game theoretic approach and reordered the affected action. Now, Filler will give its signature to FeeHub after the user initiates the swap on the origin chain. FeeHub is also bound to give signatures for Delegates as Filler won't initiate on the destination chain till the verification of all the signatures from FeeHub to Delegates. Incentive for Filler to do the same is to get matched for future orders, because if Filler didn't give a signature for one order, then the Delegates will `changeVote()` to an honest Filler.*

TOB-CATALOG-4: A creator can claim fees without redeeming the order
Further investigation required. The proposed fixes are outside the scope of this review and require further investigation to confirm resolution.

The Catalog Finance team provided the following context for this finding's fix status:

> *Took the recommendation and updated the required timelines in our off-chain system.*

TOB-CATALOG-5: Delegators can extend their stake at a higher voting multiple
Resolved. The Catalog Finance team informed us that this is an intended feature and provided the following context for this finding's fix status:

> *This is a feature, not a bug, by protocol design. Our intent is to make users stake in the system for a longer duration. It is totally fine to have a multiplier for a duration that a user already completed and extending the same stake.*

We recommend adding some documentation about this to inform future development.

TOB-CATALOG-6: Inconsistent use of the term "expiry"
Resolved in PR #3. The `expiry` variable has been replaced with the `timelock` variable in the `GardenFEEAccount` contract.

The Catalog Finance team provided the following context for this finding's fix status:

> *Updated appropriate variable name to `expiry` wherever it is referring to the block number at which it expires and `timelock` wherever it is referring to the duration of time to expire.*

TOB-CATALOG-7: Improper handling of delegateStakeIDs disables critical functionality
Resolved in PR #4. A call to the `checkRole` function was added to `getVotes`, which causes it to revert if the target filler is not active. Additionally, during the filler refund process, the entire struct is no longer deleted and `delegateStakeIDs` is intentionally left untouched. This has the effect of allowing stakers to call `changeVote` and `refund` while the filler is refunded, and if the filler re-registers, the stakers will be reactivated automatically.

The Catalog Finance team provided the following context for this finding's fix status:

> *Instead of using `delete()` to remove the filler entity from the system, we now reset all data members except `delegateStakeIDs` for the filler. Added `_checkRole()` in `getVotes()`.*

Additionally, we recommend adding documentation about this issue, including a comment about not deleting `delegateStakeIDs`, as this may present a problem during future development.

TOB-CATALOG-8: A user cannot renew their stake for the maximum duration
Resolved in PR #5. The logic has been updated to account for the maximum duration. The Catalog Finance team provided the following context for this finding's fix status:

> *Updated renew() to check for permanent stake condition.*

TOB-CATALOG-9: Delegators lose rewards when HTLCs expire before others in a channel

Resolved in PR #6. Both claimed and submitted secrets are now stored in mappings in the `GardenFEEAccount` contract. The Catalog Finance team provided the following context for this finding's fix status:

> *Updated claim() to be flexible for more secrets with same nonce it is already present in signed state by FeeManager.*

G. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Further Investigation Required	Further investigation is required to confirm the resolution as the fix may involve code that is out of scope.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.