SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Carapace
Prepared by:	Sherlock
Lead Security Expert:	clems4ever
Dates Audited:	February 9 - February 23, 2023
Prepared on:	March 10, 2023

Introduction

Building DeFi's newest primitive - connecting buyers sellers of credit default risk in crypto loans.

Scope

All contracts in `contracts` folder, excluding `test` folder.

./:

- UUPSUpgradeableBase.sol

./adapters:

- GoldfinchAdapter.sol : upgradeable using UUPS pattern

./core:

- ContractFactory.sol : upgradeable using UUPS pattern
- DefaultStateManager.sol : upgradeable using UUPS pattern
- PremiumCalculator.sol : upgradeable using UUPS pattern
- ProtectionPoolCycleManager.sol : upgradeable using UUPS pattern

./core/pool:

- **ProtectionPool.sol : This is core contract of the protocol and upgradeable using UUPS pattern**
- ReferenceLendingPools.sol : upgradeable using UUPS pattern
- **SToken.sol : ERC-20 compliant implementation of the interest bearing token for the Carapace protocol**

./external/goldfinch:

- ConfigOptions.sol
- ICreditLine.sol
- IGoldfinchConfig.sol
- IPoolTokens.sol
- ISeniorPool.sol
- ISeniorPoolStrategy.sol
- ITranchedPool.sol
- IV2CreditLine.sol

./external/openzeppelin/ERC1967:

- ERC1967Proxy.sol
- Proxy.sol

./interfaces:

- IDefaultStateManager.sol
- ILendingProtocolAdapter.sol
- ILendingProtocolAdapterFactory.sol
- IPremiumCalculator.sol
- IProtectionPool.sol
- IProtectionPoolCycleManager.sol
- IReferenceLendingPools.sol

./libraries:

- AccruedPremiumCalculator.sol
- Constants.sol
- ProtectionPoolHelper.sol
- RiskFactorCalculator.sol

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

Medium	High
8	11

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

clems4ever
0Kage
bin2chen
rvierdiiev
Koolex
mahdikarimi
Jeiwan
chaduke
jkoppel
immeas
monrel
libratus
0x52
charlesjhongc
minhtrng
XKET
KingNFT
MalfurionWhitehat
Tricko
__141345__
dec3ntraliz3d
VAD37
Web3SecurityDAO
ctf_sec
unforgiven
Allarious
modern_Alchemist_00
Kumpa
c7e7eff
Bauer
Ruhum
ast3ros
mert_eren
jprod15
peanuts
weeeh_
SPYBOY
yixxas
ktg
HonorLt
carrot
ck
csanuragjain

Issue H-1: User can game protection via renewal to get free insurance

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/293

Found by

jkoppel, 0x52, 0Kage, minhtrng, libratus, rvierdiiev, monrel, KingNFT

Summary

When renewing a position, the new protectionAmount can be higher than what was previously bought. A user can abuse this to get insurance for free. Most of the time they would keep the protection amount at 1 and pay virtually no premium. Since late payments can be seen very far in advance they would simply renew their insurance at the max value of token right before the borrower was officially late and gain the full protection.

Vulnerability Detail

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/libraries/ProtectionPoolHelper.sol#L383-L398

When evaluating if a user can renew their insurance only the time, token and lending contract are verified. It never checks the amount of protection that the user is renewing for. This can be abused to renew for MORE protection than originally bought. A user could abuse this to renew right before there was a late payment for the full amount of protection.

They can abuse another quirk of the renewal system to make sure they they are always able to renew at any time. Since a user is allowed to open an unlimited number of positions on a single LP token they can open a large number of positions with 1 protection amount. They would space out each protection to expire exactly with the grace period. The results it that they would be able to renew any position at a moments notice.

They would abuse this by choosing to renew their protection for the max value of the token right before a payment was officially late. This would allow them to collect a full repayment while paying basically nothing in premium.

Impact

User can get full protection for free

Code Snippet

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L176-L195

Tool used

Manual Review

Recommendation

When renewing protection, the user should only be allowed to renew up to the value of their expired insurance

Discussion

vnadoda

@clems4ev3r what's the basis for the following claim? How would a buyer know abt late payment in advance? "Since late payments can be seen very far in advance they would simply renew their insurance at the max value of token right before the borrower was officially late and gain the full protection." Cc @taisukemino

clems4ev3r

> @clems4ev3r what's the basis for the following claim? How would a buyer know abt late payment in advance? "Since late payments can be seen very far in advance they would simply renew their insurance at the max value of token right before the borrower was officially late and gain the full protection." Cc @taisukemino

I think this is possible due to the fact that any user can go fetch lending pool status from goldfinch directly. So they are aware of state changes not yet triggered on Carapace and can adjust their positions accordingly

vnadoda

> @clems4ev3r what's the basis for the following claim? How would a buyer know abt late payment in advance? "Since late payments can be seen very far in advance they would simply renew their insurance at the max value of token right before the borrower was officially late and gain the full protection." Cc @taisukemino
>
> I think this is possible due to the fact that any user can go fetch lending pool status from goldfinch directly. So they are aware of state changes not yet triggered on Carapace and can adjust their positions accordingly

Yeah, buyers can't see far in the future, but even early knowledge of the late payment by a couple of hours can be exploited.

We are planning to fix this issue. @taisukemino

vnadoda

@clems4ev3r PR for this fix is: https://github.com/carapace-finance/credit-default-swaps-contracts/pull/56

when you review fix PRs, please do it in sequence, they are created as I am creating a new fix branch on top of the previous fix branch to avoid merge conflicts. Cc @hrishibhat

Issue H-2: Protection sellers can bypass withdrawal delay mechanism and avoid losing funds when loans are defaulted by creating withdrawal request in each cycle

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/292

Found by

ctf_sec, Jeiwan, peanuts, mahdikarimi, HonorLt, 0Kage, csanuragjain, ktg, chaduke, jprod15, mert_eren, rvierdiiev, KingNFT, Allarious, immeas, Ruhum, jkoppel, carrot, XKET, Bauer, ck, bin2chen, ast3ros, 0x52, unforgiven, libratus, monrel, clems4ever

Summary

To prevent protection sellers from withdrawing fund immediately when protected lending pools are defaults, there is withdrawal delay mechanism, but it's possible to bypass it by creating withdraw request in each cycle by doing so user can withdraw in each cycle's open state. there is no penalty for users when they do this or there is no check to avoid this.

Vulnerability Detail

This is `_requestWithdrawal()` code:

```
function _requestWithdrawal(uint256 _sTokenAmount) internal {
  uint256 _sTokenBalance = balanceOf(msg.sender);
  if (_sTokenAmount > _sTokenBalance) {
    revert InsufficientSTokenBalance(msg.sender, _sTokenBalance);
  }

  /// Get current cycle index for this pool
  uint256 _currentCycleIndex = poolCycleManager.getCurrentCycleIndex(
    address(this)
  );

  /// Actual withdrawal is allowed in open period of cycle after next cycle
  /// For example: if request is made in at some time in cycle 1,
  /// then withdrawal is allowed in open period of cycle 3
  uint256 _withdrawalCycleIndex = _currentCycleIndex + 2;

  WithdrawalCycleDetail storage withdrawalCycle = withdrawalCycleDetails[
    _withdrawalCycleIndex
  ];

  /// Cache existing requested amount for the cycle for the sender
```

```
  uint256 _oldRequestAmount = withdrawalCycle.withdrawalRequests[msg.sender];
  withdrawalCycle.withdrawalRequests[msg.sender] = _sTokenAmount;

  unchecked {
    /// Update total requested withdrawal amount for the cycle considering
↪   existing requested amount
    if (_oldRequestAmount > _sTokenAmount) {
      withdrawalCycle.totalSTokenRequested -= (_oldRequestAmount -
        _sTokenAmount);
    } else {
      withdrawalCycle.totalSTokenRequested += (_sTokenAmount -
        _oldRequestAmount);
    }
  }

  emit WithdrawalRequested(msg.sender, _sTokenAmount, _withdrawalCycleIndex);
}
```

As you can see it doesn't keep track of user current withdrawal requests and user can request withdrawal for all of his balance in each cycle and by doing so user can set `withdrawalCycleDetails[Each Cycle][User]` to user's sToken balance. and whenever user wants to withdraw he only need to wait until the end of the current cycle while he should have waited until next cycle end.

Impact

protection sellers can request withdraw in each cycle for their full sToken balance and code would allow them to withdraw in each cycle end time because code doesn't track how much of the balance of users is requested for withdrawals in the past.

Code Snippet

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L1061-L1097

Tool used

Manual Review

Recommendation

To avoid this code should keep track of user balance that is not in withdraw delay and user balance that are requested for withdraw. and to prevent users from requesting withdrawing and not doing it protocol should have some penalties for

withdrawals, for example the waiting withdraw balance shouldn't get reward in waiting duration.

Issue H-3: Lending pool state transition will be broken when pool is expired in late state

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/230

Found by

rvierdiiev, Jeiwan

Summary

Lending pool state transition will be broken when pool is expired in late state

Vulnerability Detail

Each lending pool has its state. State is calculated inside `ReferenceLendingPools._getLendingPoolStatus` function. https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ReferenceLendingPools.sol#L318-L349

```
function _getLendingPoolStatus(address _lendingPoolAddress)
  internal
  view
  returns (LendingPoolStatus)
{
  if (!_isReferenceLendingPoolAdded(_lendingPoolAddress)) {
    return LendingPoolStatus.NotSupported;
  }


  ILendingProtocolAdapter _adapter = _getLendingProtocolAdapter(
    _lendingPoolAddress
  );


  if (_adapter.isLendingPoolExpired(_lendingPoolAddress)) {
    return LendingPoolStatus.Expired;
  }


  if (
    _adapter.isLendingPoolLateWithinGracePeriod(
      _lendingPoolAddress,
      Constants.LATE_PAYMENT_GRACE_PERIOD_IN_DAYS
    )
  ) {
```

```
    return LendingPoolStatus.LateWithinGracePeriod;
  }


  if (_adapter.isLendingPoolLate(_lendingPoolAddress)) {
    return LendingPoolStatus.Late;
  }


  return LendingPoolStatus.Active;
}
```

Pls, note, that the first state that is checked is `expired`.
https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/adapters/GoldfinchAdapter.sol#L62-L77

```
function isLendingPoolExpired(address _lendingPoolAddress)
  external
  view
  override
  returns (bool)
{
  ICreditLine _creditLine = _getCreditLine(_lendingPoolAddress);
  uint256 _termEndTimestamp = _creditLine.termEndTime();


  /// Repaid logic derived from Goldfinch frontend code:
  /// https://github.com/goldfinch-eng/mono/blob/bd9adae6fbd810d1ebb5f7ef22df5bb
↪  6f1eaee3b/packages/client2/lib/pools/index.ts#L54
  /// when the credit line has zero balance with valid term end, it is
↪  considered repaid
  return
    block.timestamp >= _termEndTimestamp ||
    (_termEndTimestamp > 0 && _creditLine.balance() == 0);
}
```

As you can see, pool is expired if time of credit line hasended or loan is fully paid.

State transition for lending pool is done inside `DefaultStateManager._assessState` function. This function is responsible to lock capital, when state is late and unlock it when it's changed from late to active again.

Because the first state that is checked is `expired` there can be few problems.

First problem. Suppose that lending pool is in late state. So capital is locked. There are 2 options now: payment was done, so pool becomes active and capital unlocked, payment was not done then pool has defaulted. But in case when state is late, and lending pool expired or loan is fully repaid(so it's also becomes expired),

then capital will not be unlocked asthereisnosuchtransitionLate->Expired. The state will be changed to Expired and no more actions will be done. Also in this case it's not possible to detect if lending pool expired because of time or because no payment was done.

Second problem. Lending pool is in active state. Last payment should be done some time before `_creditLine.termEndTime()`. Payment was not done, which means that state should be changed to Late and capital should be locked, but state was checked when loan has ended, so it became Expired and again there is no such transition that can detect that capital should be locked in this case. The state will be changed to Expired and no more actions will be done.

Impact

Depending on situation, capital can be locked forever or protection buyers will not be compensated.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

These are tricky cases, think about transition for lending pool in such cases.

Discussion

vnadoda

@clems4ev3r We are planning to fix this, possibly using recommendation mentioned in a duplicate #251

clems4ev3r

@vnadoda agreed

Issue H-4: Existing buyer who has been regularly renewing protection will be denied renewal even when she is well within the renewal grace period

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/174

Found by

0Kage

Summary

Existing buyers have an opportunity to renew their protection within grace period. If lending state update happens from `Active` to `LateWithinGracePeriod` just 1 second after a buyer's protection expires, protocol denies buyer an opportunity even when she is well within the grace period.

Since defaults are not sudden and an `Active` loan first transitions into `LateWithinGracePeriod`, it is unfair to deny an existing buyer an opportunity to renew (its alright if a new protection buyer is DOSed). This is especially so because a late loan can become `active` again in future (or move to `default`, but both possibilities exist at this stage).

All previous protection payments are a total loss for a buyer when she is denied a legitimate renewal request at the first sign of danger.

Vulnerability Detail

`renewProtection` first calls `verifyBuyerCanRenewProtection` that checks if the user requesting renewal holds same NFT id on same lending pool address & that the current request is within grace period defined by protocol.

Once successfully verified, `renewProtection` calls _verifyAndCreateProtection to renew protection. This is the same function that gets called when a new protection is created.

Notice that this function calls `_verifyLendingPoolIsActive` as part of its verification before creating new protection - this check denies protection on loans that are in `LateWithinGracePeriod` or `Late` phase (see snippet below).

```
function _verifyLendingPoolIsActive(
    IDefaultStateManager defaultStateManager,
    address _protectionPoolAddress,
    address _lendingPoolAddress
  ) internal view {
    LendingPoolStatus poolStatus = defaultStateManager.getLendingPoolStatus(
```

```
      _protectionPoolAddress,
      _lendingPoolAddress
    );

    ...
    if (
      poolStatus == LendingPoolStatus.LateWithinGracePeriod ||
      poolStatus == LendingPoolStatus.Late
    ) {
      revert IProtectionPool.LendingPoolHasLatePayment(_lendingPoolAddress);
    }
    ...
}
```

Impact

User who has been regularly renewing protection and paying premium to protect against a future loss event will be denied that very protection when she most needs it.

If existing user is denied renewal, she can never get back in (unless the lending pool becomes active again). All her previous payments were a total loss for her.

Code Snippet

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L189

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/libraries/ProtectionPoolHelper.sol#L407

Tool used

Manual Review

Recommendation

When a user is calling `renewProtection`, a different implementation of `verifyLendingPoolIsActive` is needed that allows a user to renew even when lending pool status is `LateWithinGracePeriod` or `Late`.

Recommend using `verifyLendingPoolIsActiveForRenewal` function in renewal flow as shown below

```
function verifyLendingPoolIsActiveForRenewal(
  IDefaultStateManager defaultStateManager,
```

```
    address _protectionPoolAddress,
    address _lendingPoolAddress
  ) internal view {
    LendingPoolStatus poolStatus = defaultStateManager.getLendingPoolStatus(
      _protectionPoolAddress,
      _lendingPoolAddress
    );

    if (poolStatus == LendingPoolStatus.NotSupported) {
      revert IProtectionPool.LendingPoolNotSupported(_lendingPoolAddress);
    }
    //------ audit - this section needs to be commented-----//
    //if (
    //  poolStatus == LendingPoolStatus.LateWithinGracePeriod ||
    //  poolStatus == LendingPoolStatus.Late
    //) {
    //  revert IProtectionPool.LendingPoolHasLatePayment(_lendingPoolAddress);
    //}
    // ------------------------------------------------------//

    if (poolStatus == LendingPoolStatus.Expired) {
      revert IProtectionPool.LendingPoolExpired(_lendingPoolAddress);
    }

    if (poolStatus == LendingPoolStatus.Defaulted) {
      revert IProtectionPool.LendingPoolDefaulted(_lendingPoolAddress);
    }
  }
```

Discussion

vnadoda

@clems4ev3r, @taisukemino and I will discuss this internally

vnadoda

@clems4ev3r planning to fix this.

clems4ev3r

@vnadoda yes this is a valid issue

Issue H-5: Protection seller will lose unlocked capital if it fails to claim during more than one period

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/142

Found by

Tricko, jkoppel, MalfurionWhitehat, XKET, Koolex, bin2chen, VAD37, dec3ntraliz3d, immeas

Summary

The protection seller will lose unlocked capital if it fails to claim during more than one period.

Vulnerability Detail

The function `DefaultStateManager._calculateClaimableAmount`, used by `DefaultStateManager.calculateAndClaimUnlockedCapital`, which in turn is used by `ProtectionPool.claimUnlockedCapital`, overridestheclaimableunlockedcapital on every loop iteration on the lockedCapitalsarray.

As a result, onlythelastsnapshotisreturned by this function, regardless if the protection seller has claimed the unlocked capital or not. The purpose of this code was to prevent sellers from claimingthesamesnapshottwice, but since the `_claimableUnlockedCapital` variable is being overwritten instead of incremented, on each loop iteration, it will also make sellers lose unlocked capital if they fail to claim at each snapshot.

Proof of concept:

1. Pool goes to locked state with snapshotId 1
2. Pool goes to active state
3. Pool goes to locked state with snapshotId 2
4. Pool goes to active state
5. Protection seller calls `ProtectionPool.claimUnlockedCapital`, but they will only receive what's due from snapshotId 2, not from snapshotId 1

Impact

The protection seller will lose unlocked capital if it fails to claim during more than one period.

Code Snippet

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/DefaultStateManager.sol#L500-L505

Tool used

Manual Review

Recommendation

Increment `_claimableUnlockedCapital` forall locked capital instances.

Issue H-6: Sybil on withdrawal requests can allow leverage factor manipulation with flashloans

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/116

Found by

mahdikarimi, clems4ever

Summary

To be able to withdraw, a user has to request a withdraw first. The only requirement to be able to request a withdraw is to have a balance of SToken upon requesting. By requesting withdraws with the same tokens but from different addresses, a malicious user can create the option to withdraw during one cycle more than what is deposited in the protocol. They cannot drain the protocol since they only have a limited amount of SToken to burn (required to call `withdraw()`), but they acquire the ability to deposit new funds and withdraw them in the same block, thus manipulating premium prices.

Vulnerability Detail

Consider the following scenario: A malicious user wants to manipulate `leverageRatio` (to get a cheaper premium for example).

They deposit 10k USDC into the protocol, and get 10k STokens. They request immediately a withdraw, and transfer STokens to another address and request a withdraw there, repeating the process 10 times.

This works since balance is checked on requesting withdrawal but not locked or committed: https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L992-L995

2 cycles later (actually ~1 cycle if the timing is optimized), they have the ability to take a flashloan for 100k USDC, deposit through the 10 addresses used, enjoy the cheaper premium as a protection buyer due to `leverageFactor` being high and withdraw all in the same transaction.

They can safely repay the flash loan.

Impact

Protection buyers can use this to:

- game premium prices, meaning that protection sellers get rugged.

- overprotect their lending positions (used in conjunction with HIGH-02, it can drain the whole protection pool if lending pool defaults).
- DOS the protocol by sending the leverage factor very high.

Code Snippet

Tool used

Manual Review

Recommendation

Freeze STokens for a depositor once they requested a withdrawal.

Issue H-7: Protection buyer may buy multiple protections for same goldfinch NFT

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/112

Found by

ctf_sec, jkoppel, 0x52, __141345__, 0Kage, modern_Alchemist_00, minhtrng, libratus, c7e7eff, Allarious, bin2chen, clems4ever, chaduke, immeas

Summary

The Carapace protocol checks that a protection buyer does not buy a protection for an amount greater than the remainingPrincipal in the corresponding loan. However it possible for the buyer to buy multiple different protections for the same Goldfinch loan.

Vulnerability Detail

The check for the possibility for a user to buy a protection is done here in `ReferenceLendingPools.canBuyProtection`: https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ReferenceLendingPools.sol#L132-L168

It checks the protection about to be created does not cross remaining principal. But it still allows the user to create multiple protections for the same loan position.

Impact

The malicious user can `overprotect` their loan position on Goldfinch and thus claim a larger amount on loan default than what they lended. For now as the default claiming feature is not implemented, they can use this bug to DOS the protocol by using all funds deposited into the protocol reaching `leverageRatioFloor` and not allowing any new protections to be bought.

Code Snippet

Tool used

Manual Review

Recommendation

Keep track of the total protection subscribed for a given loan and limit total protection value to remaining capital

Discussion

vnadoda

@clems4ev3r this is duplicate of #193 & #139

hrishibhat

Sponsor comment from #193:

Double buying of protections for the same NFT is a known issue and we were planning to tackle it in an upcoming version because even after buying multiple protections buyers won't be able to claim for the same position as default payout will require NFT lock/transfer in the carapace vault.

This double counting of locked capital issue seems a legit concern. Now we are considering fixing this with other audit issues.

Issue H-8: Too many active protections can cause the ProtectionPool reach the block gas limit

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/63

Found by

ctf_sec, peanuts, SPYBOY, __141345__, 0Kage, chaduke, rvierdiiev, KingNFT, yixxas, Ruhum, jkoppel, weeeh_, Bauer, modern_Alchemist_00, bin2chen, Koolex, Tricko, ast3ros, MalfurionWhitehat, 0x52, unforgiven, minhtrng, libratus, clems4ever

Summary

There are two instances where the ProtectionPool contract loops over an unbounded array. These can cause the transaction to succeed the block gas limit causing the transaction to revert, see https://swcregistry.io/docs/SWC-128.

Vulnerability Detail

Both in `lockCapital()` and in `_accruePremiumAndExpireProtections()` the contract loops over the unbounded array of protections. If there are too many protections the transaction will revert because it reached the block gas limit.

Impact

Both the `lockCapital()` and `_accruePremiumAndExpireProtections()` functions are critical components of the contract. Them not being accessible renders the contract useless. Protection buyers won't be covered in case of the underlying pool defaulting because the deposited tokens can't be locked up

Code Snippet

`lockCapital()`: https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L378-L411

`_accruePremiumAndExpireProtections()`: https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L957-L1021

Tool used

Manual Review

Recommendation

You either have to limit the number of protections so that it is impossible that you surpass the block gas limit. Or, you change the logic so that you're never forced to loop over all the existing protections.

Discussion

vnadoda

@clems4ev3r We are aware of this concern re: `accruePremiumAndExpireProtections` and hence we had put mitigation in place. If there are too many protections to iterate over, we can call the function `accruePremiumAndExpireProtections` in batch using `_lendingPools` param. See https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L279

The function `lockCapital` deals with one lending pool at a time, so practically it will not have too many protections to iterate over.

clems4ev3r

@vnadoda `activeProtectionIndexes` is unbounded even for one lendingPool. Any malicious user can stuff the array by taking small protections and cause the DOS. This sounds like a valid issue

vnadoda

@clems4ev3r @taisukemino let's discuss possible remedies. can we set up conf call?

vnadoda

@clems4ev3r As discussed on the call, the following 2 fixes should reduce `lockCapital` attack vector significantly:

1. Don't allow buyers to buy protection for the same NFT/lending position multiple times
2. Introduce min protection amount param

@taisukemino let's discuss this internally.

vnadoda

@hrishibhat we are planning to address this

Issue H-9: Missing validation of snapshotId makes it possible for the investor to claim unlocked capitals from the same snapshot multiple times

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/60

Found by

bin2chen, Koolex

Summary

Missing validation of snapshotId makes it possible for the investor/seller to claim unlocked capitals from the same snapshot multiple times

Vulnerability Detail

Description

The seller can call `ProtectionPool.claimUnlockedCapital` function to claim unlocked capitals. It then calls `defaultStateManager.calculateAndClaimUnlockedCapital` to calculate the claimable amount, then it transfers the amount to the seller if it is greater than zero. `defaultStateManager.calculateAndClaimUnlockedCapital` function works as follows:

1. Iterates over all lending pools.
2. For each lending pool calculates the claimable amount for the seller.
3. Updates the last claimed snapshot id for the seller for the given lending pool so it becomes uncalimable next time.

However, before updating the last claimed snapshot id, it doesn't check if the returned snapshot Id (from `_calculateClaimableAmount` function) is zero. This means if it happens that the returned value is zero, the last claimed snapshot id will be reset to its initial value (zero) and the seller can claim again as if s/he never did.

PoC

Given: A pool protection with one lending pool for simplicity Seller's lastClaimedSnapshotId = 0

Imagine the following sequence of events:

- **Event**: Lending pool transition from **Active to Late**
  - A snapshot taken

    - LockedCapitals[0].snapshotId = 1
    - LockedCapitals[0].locked = true
- **Event**: Lending pool transition from **Late to Active**
    - LockedCapitals[0].snapshotId = 1
    - LockedCapitals[0].locked = false
- **Event**: **Seller claims** => receives his/her share of the unlocked capital
    - LastClaimedSnapshotId = 1
- **Event**: **Seller claims** => doesn't receive anything
    - LastClaimedSnapshotId = 0 (this is the issue as it is reset to zero)
- **Now the seller can claim again from the same snapshot**
- **Event**: **Seller claims** => receives his/her share of the unlocked capital
    - LastClaimedSnapshotId = 1

This can be repeated till all funds/capitals are drained.

This happens when all snapshots were claimed before, then the function `_calculateClaimableAmount` will return_latestClaimedSnapshotId as zero.

Impact

- An investor/seller could possibly claim unlocked capitals from the same snapshot multiple times which is unfair.
- An attacker could join the pool as seller, later drains the funds from the protection pool whenever any lending pool goes into Late then Active state again.

Code Snippet

- ProtectionPool.claimUnlockedCapital

```
/// Investors can claim their total share of released/unlocked capital across
 ↪  all lending pools
uint256 _claimableAmount = defaultStateManager
  .calculateAndClaimUnlockedCapital(msg.sender);

if (_claimableAmount > 0) {
  console.log(
    "Total sToken underlying: %s, claimableAmount: %s",
    totalSTokenUnderlying,
    _claimableAmount
  );
```

```
  /// transfer the share of unlocked capital to the receiver
  poolInfo.underlyingToken.safeTransfer(_receiver, _claimableAmount);
}
```

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L427-L445

- defaultStateManager.calculateAndClaimUnlockedCapital

```
/// Calculate the claimable amount across all the locked capital instances for a
 ↪  given protection pool
    (
      uint256 _unlockedCapitalPerLendingPool,
      uint256 _snapshotId
    ) = _calculateClaimableAmount(poolState, _lendingPool, _seller);
    _claimedUnlockedCapital += _unlockedCapitalPerLendingPool;

    /// update the last claimed snapshot id for the seller for the given
     ↪  lending pool,
    /// so that the next time the seller claims, the calculation starts from
     ↪  the last claimed snapshot id
    poolState.lastClaimedSnapshotIds[_lendingPool][_seller] = _snapshotId;
```

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/DefaultStateManager.sol#L186-L195

- defaultStateManager._calculateClaimableAmount

```
/// Verify that the seller does not claim the same snapshot twice
if (!lockedCapital.locked && _snapshotId > _lastClaimedSnapshotId) {
  ERC20SnapshotUpgradeable _poolSToken = ERC20SnapshotUpgradeable(
    address(poolState.protectionPool)
  );

  console.log(
    "balance of seller: %s, total supply: %s at snapshot: %s",
    _poolSToken.balanceOfAt(_seller, _snapshotId),
    _poolSToken.totalSupplyAt(_snapshotId),
    _snapshotId
  );

  /// The claimable amount for the given seller is proportional to the seller's
   ↪  share of the total supply at the snapshot
  /// claimable amount = (seller's snapshot balance / total supply at snapshot)
 ↪  * locked capital amount
  _claimableUnlockedCapital =
    (_poolSToken.balanceOfAt(_seller, _snapshotId) *
```

```
      lockedCapital.amount) /
    _poolSToken.totalSupplyAt(_snapshotId);

  /// Update the last claimed snapshot id for the seller
  _latestClaimedSnapshotId = _snapshotId;

  console.log(
    "Claimable amount for seller %s is %s",
    _seller,
    _claimableUnlockedCapital
  );
}
```

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/DefaultStateManager.sol#L487-L515

Tool used

Manual Review

Recommendation

Only update the last snapshot id of the seller if it is greater than zero.

Example:

```
if(_snapshotId > 0){
      poolState.lastClaimedSnapshotIds[_lendingPool][_seller] = _snapshotId;
    }
```

Discussion

vnadoda

@clems4ev3r this isn't a valid concern/issue. The function `DefaultStateManager._calculateClaimableAmount` always starts with the seller's last claimed snapshot id and not with 0. So in the scenario/PoC mentioned, when the seller tries to claim a second time, returned `_latestClaimedSnapshotId` will be `1` and not 0.

See: https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/DefaultStateManager.sol#L466

clems4ev3r

@vnadoda surprisingly this seems valid.

After a user has claimed for a snapshot, latest snapshot claimed is correctly set here: https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/DefaultStateManager.sol#L195

If the user calls the function again, they will be able to claim zero funds since the condition here is always false: https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/DefaultStateManager.sol#L488

Unfortunately that means the variable `_latestClaimedSnapshotId` is not set here: https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/DefaultStateManager.sol#L508

and so `_calculateClaimableAmount` returns a `_latestClaimedSnapshotId == 0` resetting latest claimed snapshot and user can claim again

vnadoda

@clems4ev3r ah, I see it now. var at the beginning is `_lastClaimedSnapshotId` is different than return var `_latestClaimedSnapshotId`. I will add this to the fix list

vnadoda

@clems4ev3r PR for this fix is: https://github.com/carapace-finance/credit-default-swaps-contracts/pull/55

Issue H-10: Malicious seller forced break lockCapital()

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/31

Found by

bin2chen, clems4ever

Summary

Malicious burn nft causes failure to lockCapital() seller steady earn PremiumAmount, buyer will be lost compensation

Vulnerability Detail

When the status of the lendingPool changes from Active to Late, the protocol will call ProtectionPool.lockCapital() to lock amount lockCapital() will loop through the active protections to calculate the `lockedAmount`. The code is as follows:

```
  function lockCapital(address _lendingPoolAddress)
    external
    payable
    override
    onlyDefaultStateManager
    whenNotPaused
    returns (uint256 _lockedAmount, uint256 _snapshotId)
  {
....
    uint256 _length = activeProtectionIndexes.length();
    for (uint256 i; i < _length; ) {
...
      uint256 _remainingPrincipal = poolInfo
        .referenceLendingPools
        .calculateRemainingPrincipal(           //<----------- calculate
↪  Remaining Principal
          _lendingPoolAddress,
          protectionInfo.buyer,
          protectionInfo.purchaseParams.nftLpTokenId
        );
```

The important thing inside is to calculate the _remainingPrincipal by `referenceLendingPools.calculateRemainingPrincipal()`

```
  function calculateRemainingPrincipal(
    address _lendingPoolAddress,
    address _lender,
```

```
    uint256 _nftLpTokenId
  ) public view override returns (uint256 _principalRemaining) {
...

    if (_poolTokens.ownerOf(_nftLpTokenId) == _lender) {   //<------------call
↪   ownerOf()
      IPoolTokens.TokenInfo memory _tokenInfo = _poolTokens.getTokenInfo(
        _nftLpTokenId
      );

....
      if (
        _tokenInfo.pool == _lendingPoolAddress &&
        _isJuniorTrancheId(_tokenInfo.tranche)
      ) {
        _principalRemaining =
          _tokenInfo.principalAmount -
          _tokenInfo.principalRedeemed;
      }
    }
  }
```

GoldfinchAdapter.calculateRemainingPrincipal() The current implementation will first determine if the ownerOf the NFTID is _lender

There is a potential problem here, if the NFTID has been burned, the ownerOf() will be directly revert, which will lead to calculateRemainingPrincipal() revert,and lockCapital() revert and can't change status from active to late

Let's see whether Goldfinch's implementation supports burn(NFTID), and whether ownerOf(NFTID) will revert

1. PoolTokens has burn() method , if principalRedeemed==principalAmount you can burn it

```
contract PoolTokens is IPoolTokens, ERC721PresetMinterPauserAutoIdUpgradeSafe,
↪   HasAdmin, IERC2981 {
.....
  function burn(uint256 tokenId) external virtual override whenNotPaused {
    TokenInfo memory token = _getTokenInfo(tokenId);
    bool canBurn = _isApprovedOrOwner(_msgSender(), tokenId);
    bool fromTokenPool = _validPool(_msgSender()) && token.pool == _msgSender();
    address owner = ownerOf(tokenId);
    require(canBurn || fromTokenPool, "ERC721Burnable: caller cannot burn this
↪   token");
    require(token.principalRedeemed == token.principalAmount, "Can only burn
↪   fully redeemed tokens");
    _destroyAndBurn(tokenId);
```

```
    emit TokenBurned(owner, token.pool, tokenId);
  }
```

https://github.com/goldfinch-eng/mono/blob/88f0e3f94f6dd23ebae429fe09e2511650df893a/packages/protocol/contracts/protocol/core/PoolTokens.sol#L199

2.ownerOf() if nftid don't exists will revert with message "ERC721: owner query for nonexistent token"

```
contract ERC721UpgradeSafe is
  Initializable,
  ContextUpgradeSafe,
  ERC165UpgradeSafe,
  IERC721,
  IERC721Metadata,
  IERC721Enumerable
{
...
  function ownerOf(uint256 tokenId) public view override returns (address) {
    return _tokenOwners.get(tokenId, "ERC721: owner query for nonexistent
↪   token");
  }
```

https://github.com/goldfinch-eng/mono/blob/88f0e3f94f6dd23ebae429fe09e2511650df893a/packages/protocol/contracts/external/ERC721.sol#L136-L138

If it can't changes to late, Won't lock the fund, seller steady earn PremiumAmount

So there are two risks

1. normal buyer gives NFTID to burn(), he does not know that it will affect all protection of the lendingPool
2. Malicious seller can buy a protection first, then burn it, so as to force all protection of the lendingPool to expire and get the PremiumAmount maliciously. buyer unable to obtain compensation

Suggested try catch for _poolTokens.ownerOf() If revert, it is assumed that the lender is not the owner

Impact

buyer will be lost compensation

Code Snippet

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L389-L395

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/adapters/GoldfinchAdapter.sol#L162-L165

Tool used

Manual Review

Recommendation

try catch for _poolTokens.ownerOf() If revert, it is assumed that the lender is not the owner

Issue H-11: Sandwich attack to accruePremiumAndExpireProtections()

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/26

Found by

immeas, monrel, chaduke, 0Kage

Summary

`accruePremiumAndExpireProtections()` will increase `totalSTokenUnderlying`, and thus increase the exchange rate of the `ProtectionPool`. A malicious user can launch a sandwich attack and profit. This violates the `Fair Distribution` principle of the protocol: https://www.carapace.finance/WhitePaper#premium-pricing

Vulnerability Detail

Let's show how a malicious user, Bob, can launch a sandwich attack to `accruePremiumAndExpireProtections()` and profit.

1. Suppose there are 1,000,000 underlying tokens for the `ProtectionPool`, and `totalSupply = 1,000,000`, therefore the exchange rate is 1/1 share. Suppose Bob has 100,000 shares.
2. Suppose `accruePremiumAndExpireProtections()` is going to be called and add 100,000 to `totalSTokenUnderlying` at L346.

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L279-L354

3) Bob front-runs `accruePremiumAndExpireProtections()` and calls `deposit()` to deposit 100,000 underlying tokens into the contract. The check for `ProtectionPoolPhase` will pass for an open phase. As a result, there are 1,100,000 underlying tokens, and 1,100,000 shares, the exchange rate is still 1/1 share. Bob now has 200,000 shares.

```
function deposit(uint256 _underlyingAmount, address _receiver)
    external
    override
    whenNotPaused
    nonReentrant
  {
    _deposit(_underlyingAmount, _receiver);
  }

  function _deposit(uint256 _underlyingAmount, address _receiver) internal {
```

```
    /// Verify that the pool is not in OpenToBuyers phase
    if (poolInfo.currentPhase == ProtectionPoolPhase.OpenToBuyers) {
      revert ProtectionPoolInOpenToBuyersPhase();
    }

    uint256 _sTokenShares = convertToSToken(_underlyingAmount);
    totalSTokenUnderlying += _underlyingAmount;
    _safeMint(_receiver, _sTokenShares);
    poolInfo.underlyingToken.safeTransferFrom(
      msg.sender,
      address(this),
      _underlyingAmount
    );

    /// Verify leverage ratio only when total capital/sTokenUnderlying is higher
↪  than minimum capital requirement
    if (_hasMinRequiredCapital()) {
      /// calculate pool's current leverage ratio considering the new deposit
      uint256 _leverageRatio = calculateLeverageRatio();

      if (_leverageRatio > poolInfo.params.leverageRatioCeiling) {
        revert ProtectionPoolLeverageRatioTooHigh(_leverageRatio);
      }
    }

    emit ProtectionSold(_receiver, _underlyingAmount);
}
```

4) Now `accruePremiumAndExpireProtections()` gets called and 100,000 is added to `totalSTokenUnderlying` at L346. As a result, we have 1,200,000 underlying tokens with 1,100,000 shares. The exchange rate becomes 12/11 share.
5) Bob calls the `withdraw()` function (assume he made a request two cycles back, he could do that since he had 100,000 underlying tokens in the pool) to withdraw 100,000 shares and he will get `100,000*12/11 = 109,090` underlying tokens. So he has a profit of 9,090 underlying tokens by the sandwich attack.

Impact

A malicious user can launch a sandwich attack to `accruePremiumAndExpireProtections()` and profit.

Code Snippet

See above

Tool used

VScode

Manual Review

Recommendation

- Create a new contract as a temporary place to store the accrued premium, and then deliver it to the `ProtectionPool` over a period of time (delivery period) with some `premiumPerSecond` to lower the incentive of a quick profit by sandwich attack.
- Restrict the maximum deposit amount for each cycle.
- Restrict the maximum withdraw amount for each cycle.

Discussion

vnadoda

@clems4ev3r this is a duplicate of #294 and #204

vnadoda

@clems4ev3r can you verify this is a duplicate? @hrishibhat can we close this?

clems4ev3r

@vnadoda agreed, this is a duplicate of #294 and #204

vnadoda

@hrishibhat please close this.

Issue M-1: The renewal grace period gives users insurance for no premium

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/308

Found by

0x52, monrel, libratus, jkoppel

Summary

When a protection position is renewed, the contract checks that the expired timestamp is within the grace period of the current timestamp. The issue is that when it is renewed, it starts insurance at block.timestamp rather than the expiration of the previous protection. The result is that the grace period is effectively free insurance for the user.

Vulnerability Detail

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/libraries/ProtectionPoolHelper.sol#L390-L397

When checking if a position can be renewed it checks the expiration of the previous protection to confirm that it is being renewed within the grace period.

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L181-L194

After checking if the protection can be removed it starts the insurance at block.timestamp. The result is that the grace period doesn't collect any premium for it's duration. To abuse this the user would keep renewing at the end of the grace period for the shortest amount of time so that they would get the most amount of insurance for free.

One might argue that the buyer didn't have insurance during this time but protection can be renewed at any time during the grace period and late payments are very easy to see coming (i.e. if the payment is due in 30 days and it's currently day 29). The result is that even though *technically* there isn't insurance the user is still basically insured because they would always be able to renew before a default.

Impact

Renewal grace period can be abused to get free insurance

Code Snippet

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L176-L195

Tool used

ChatGPT

Recommendation

When renewing protection, the protection should renew from the end of the expired protection not block.timestamp.

Discussion

clems4ev3r

looks like a duplicate of #179

vnadoda

@clems4ev3r actually this is a duplicate of #190

clems4ev3r

@vnadoda agreed, as per my comment on #190: #190 #308 and #179 are duplicates

Issue M-2: function lockCapital() doesn't filter the expired protections first and code may lock more funds than required and expired defaulted protections may funded

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/305

Found by

Web3SecurityDAO, XKET, unforgiven, __141345__, rvierdiiev

Summary

when a lending loan defaults, then function `lockCapital()` get called in the ProtectionPool to lock required funds for the protections bought for that lending pool, but code doesn't filter the expired protections first and they may be expired protection in the active protection array that are not excluded and this would cause code to lock more fund and pay fund for expired defaulted protections and protection sellers would lose more funds.

Vulnerability Detail

This `lockCapital()` code:

```
function lockCapital(address _lendingPoolAddress)
  external
  payable
  override
  onlyDefaultStateManager
  whenNotPaused
  returns (uint256 _lockedAmount, uint256 _snapshotId)
{
  /// step 1: Capture protection pool's current investors by creating a snapshot
 ↪  of the token balance by using ERC20Snapshot in SToken
  _snapshotId = _snapshot();

  /// step 2: calculate total capital to be locked
  LendingPoolDetail storage lendingPoolDetail = lendingPoolDetails[
    _lendingPoolAddress
  ];

  /// Get indexes of active protection for a lending pool from the storage
  EnumerableSetUpgradeable.UintSet
    storage activeProtectionIndexes = lendingPoolDetail
      .activeProtectionIndexes;
```

```
  /// Iterate all active protections and calculate total locked amount for this
↪   lending pool
  /// 1. calculate remaining principal amount for each loan protection in the
↪   lending pool.
  /// 2. for each loan protection, lockedAmt = min(protectionAmt,
↪   remainingPrincipal)
  /// 3. total locked amount = sum of lockedAmt for all loan protections
  uint256 _length = activeProtectionIndexes.length();
  for (uint256 i; i < _length; ) {
    /// Get protection info from the storage
    uint256 _protectionIndex = activeProtectionIndexes.at(i);
    ProtectionInfo storage protectionInfo = protectionInfos[_protectionIndex];

    /// Calculate remaining principal amount for a loan protection in the
↪   lending pool
    uint256 _remainingPrincipal = poolInfo
      .referenceLendingPools
      .calculateRemainingPrincipal(
        _lendingPoolAddress,
        protectionInfo.buyer,
        protectionInfo.purchaseParams.nftLpTokenId
      );

    /// Locked amount is minimum of protection amount and remaining principal
    uint256 _protectionAmount = protectionInfo
      .purchaseParams
      .protectionAmount;
    uint256 _lockedAmountPerProtection = _protectionAmount <
      _remainingPrincipal
      ? _protectionAmount
      : _remainingPrincipal;

    _lockedAmount += _lockedAmountPerProtection;

    unchecked {
      ++i;
    }
  }

  unchecked {
    /// step 3: Update total locked & available capital in storage
    if (totalSTokenUnderlying < _lockedAmount) {
      /// If totalSTokenUnderlying < _lockedAmount, then lock all available
↪   capital
      _lockedAmount = totalSTokenUnderlying;
      totalSTokenUnderlying = 0;
    } else {
      /// Reduce the total sToken underlying amount by the locked amount
```


```
        totalSTokenUnderlying -= _lockedAmount;
      }
    }
  }
```

As you can see code loops through active protection array for that lending pool and calculates required locked amount but it doesn't call `_accruePremiumAndExpireProtections()` to make sure active protections doesn't include any expired protections. if function `_accruePremiumAndExpireProtections()` doesn't get called for a while, then there would be possible that some of the protections are expired and they are still in the active protection array. This would cause code to calculated more locked amount and also pay fund for those expired defaulted protections too from protection sellers. (also when calculating the required token payment for the protection code doesn't check the expiration too in the other functions that are get called by the `lockCapital()`, the expire check doesn't exists in inner function too)

Impact

see summery

Code Snippet

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L357-L411

Tool used

Manual Review

Recommendation

call `_accruePremiumAndExpireProtections()` for the defaulted pool to filter out the expired protections.

Discussion

vnadoda

@clems4ev3r we plan to fix this issue

Issue M-3: Protection can be bought in late pools, allowing buyers to pay minimal premium and increase the chance of a compensation

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/252

Found by

ctf_sec, Jeiwan, ast3ros, mahdikarimi, Kumpa, 0Kage, libratus, monrel, Allarious

Summary

A buyer can buy a protection for a pool that's already late on a payment. The buyer can pay the minimal premium and get a higher chance of getting a compensation. Protection sellers may bear higher losses due to reduced premium amounts and the increased chance of protection payments.

Vulnerability Detail

The protocol allows lenders on Goldfinch to get an insurance on the funds they lent. The insurance is paid after arepaymentwaslate. The protocol doesn'tallowprotectionbuyerstobuyprotectionsforpoolsthat'salreadylate to disallow buyers abusing the protections payment mechanism. To do this, the `_verifyLendingPoolIsActive` function checksthecurrentstatusofapool and revertsifit'slate.

However, `poolStatus` is cached and can be outdated when the function is called, since it's not updated in the call. Pool statuses are updated in assessStates and assessStateBatch, which are triggered on schedule separately. This allows buyers to buy protections in pools that's already late in Goldfinch but still active in Carapace.

Consider this scenario:

1. A pool is in the active state after `assessStates` is run.
2. Before the next `assessStates` call, the pool gets into the late state, due to a missed repayment. However, in the protocol, the pool is still in the active state since `assessStates` hasn't been called.
3. The malicious buyer front runts the next `assessStates` call and submits their transactions that buys a protection with the minimal duration for the pool. The `_verifyLendingPoolIsActive` function passes because the pool's state hasn't been updated in the contracts yet.
4. The `assessStates` call changes the status of the pool to `LateWithinGracePeriod`, which disallows buying protections for the pool.

5. If the pool eventually gets into the default state (chances of that is higher since there's already a late payment), the malicious buyer will be eligible for a compensation.

Impact

Protection buyers can increase their chances of getting a compensation, while buying protections with the minimal duration and paying the minimal premium. Protection sellers will bear increased loses due to reduced premium amounts and the increased chance of a compensation.

Code Snippet

1. `_verifyLendingPoolIsActive` checks the current status of a pool and reverts if it's not active: ProtectionPoolHelper.sol#L412-L415
2. Pool statuses are cached and are stored in `DefaultStateManager`: DefaultStateManager.sol#L278-L280
3. Pool statuses are updated in `DefaultStateManager.assessStates`: DefaultStateManager.sol#L119
4. `DefaultStateManager.assessStates` is not called by `ProtectionPool.buyProtection`: ProtectionPool.sol#L162

Tool used

Manual Review

Recommendation

In `ProtectionPoolHelper._verifyLendingPoolIsActive`, consider calling `DefaultStateManager._assessState` to update the status of the pool for which a protection is bought.

Issue M-4: Protection too expensive when some capital is locked

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/208

Found by

jkoppel

Summary

The leverage ratio is computed as a ratio of unlocked capital to total protections. This means that each protection position which has locked capital is effectively counted twice: once to lock up capital, and once again to decrease the leverage ratio. This can lead to situations where a pool is very well capitalized to sell more protection, but cannot do so.

Vulnerability Detail

Consider: Pool has $1.1 million in deposits, and $1 million in protection positions. All pools have late payments, and so $1 million is locked. Pool now has $100k in unallocated deposits, but its leverage ratio is $100k/1.1M \sim 0.09$. Depending on the leverage floor, protection will either be really expensive or cannot be bought at all, even though the pool is well-capitalized.

Compare: If all pools defaulted and the $1M was lost, then the pool would have $100k in deposits and $0 in protection positions, and protection would be very cheap. The situation with locked capital should be treated similarly.

Impact

Protection is very expensive or impossible in some situations where it should be cheap.

Code Snippet

Note that lockCapital() decreases totalSTokenUnderlying but does not modify totalProtections. These are the two variables used to compute the leverage ratio. See https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L357

Tool used

Manual Review

Recommendation

Do not count protections which are locking up capital when computing the leverage ratio.

Discussion

vnadoda

@taisukemino Please review this issue and we can discuss it.

vnadoda

@clems4ev3r we plan to fix this issue

clems4ev3r

@vnadoda agreed this is valid. Active protections which have contributed to locking capital should not be accounted for in leverage ratio

Issue M-5: secondary markets are problematic with how `lockCapital` works

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/147

Found by

charlesjhongc, immeas

Summary

Seeing that a pool is about to lock, an attacker can use a flash loan from a secondary market like uniswap to claim the share of a potential unlock of capital later.

Vulnerability Detail

The timestamp a pool switches to Late can be predicted and an attacker can use this to call `assessState` which is callable by anyone. This will trigger the pool to move from `Active`/`LateWithinGracePeriod` to `Late` calling `lockCapital` on the `ProtectionPool`:

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L365-L366

```
File: ProtectionPool.sol

365:     /// step 1: Capture protection pool's current investors by creating a
 ↪  snapshot of the token balance by using ERC20Snapshot in SToken
366:     _snapshotId = _snapshot();
```

This records who is holding sTokens at this point in time. If the borrower makes a payment and the pool turns back to Active, later the locked funds will be available to claim for the sToken holders at that snapshot:

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/DefaultStateManager.sol#L500-L505

```
File: DefaultStateManager.sol

500:         /// The claimable amount for the given seller is proportional to the
 ↪  seller's share of the total supply at the snapshot
501:         /// claimable amount = (seller's snapshot balance / total supply at
 ↪  snapshot) * locked capital amount
502:         _claimableUnlockedCapital =
503:           (_poolSToken.balanceOfAt(_seller, _snapshotId) *
```

```
504:             lockedCapital.amount) /
505:           _poolSToken.totalSupplyAt(_snapshotId);
```

From docs:

> If sellers wish to redeem their capital and interest before the lockup period, they might be able to find a buyer of their sToken in a secondary market like Uniswap. Traders in the exchanges can long/short sTokens based on their opinion about the risk exposure associated with sTokens. Since an sToken is a fungible ERC20 token, it is fairly easy to bootstrap the secondary markets for protection sellers.

If there is a uniswap (or similar) pool for this sToken, an attacker could potentially, using a flash loan, trigger the switch to `Late` and since they will be the ones holding the sTokens at the point of locking they will be the ones that can claim the funds at a potential unlock.

Impact

An attacker can, using a flash loan from a secondary market like uniswap, steal a LPs possible share of unlocked tokens. Only paying taking the risk of the flash loan fee.

Code Snippet

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L357-L366

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/DefaultStateManager.sol#L119

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/DefaultStateManager.sol#L137

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/DefaultStateManager.sol#L503

Tool used

Manual Review

Recommendation

I recommend you make `assessState` only callable by a trusted user. This would remove the attack vector, since you must hold the tokens over a transaction. It would still be possible to use the withdrawbug, but if that is fixed this would remove the possibility to "flash-lock".

Issue M-6: Growing of totalSupply after successive lock/unlockCa can freeze protection pools by uint overflow

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/118

Found by

clems4ever

Summary

In a protection pool, after enough cycles of locking capital/depositing, totalSupply can grow to overflow uint256.

Vulnerability Detail

In `convertToSToken`: https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L589-L606

`_getExchangeRate()` can become arbitrarily small after a funds locking, since locked funds are substracted from `totalSTokenUnderlying`; This means that new depositors can get a lot more shares than depositors from before funds locking. This behavior is correct, because otherwise previous depositors would have an oversized share of the new capital. However this has the negative effect of growing `totalSupply` exponentially, eventually reaching `type(uint).max` and overflowing (reverting every new deposit).

Impact

Protocol can come to a halt if totalSupply reaches `type(uint).max`.

Code Snippet

Tool used

Manual Review

Recommendation

Design the token in a way that it can be rebased regularly.

Discussion

vnadoda

@clems4ev3r Technically it is possible but I don't think this can happen in practice. Cc @taisukemino

clems4ev3r

@vnadoda actually the risk here is to have `_getExchangeRate() == 0` after a few lock events. If the protection pool contains 1M USDC (10**12), and a locking event leaves a dust amount in the pool (let's say 1 wei), next deposits for 1M USDC will multiply totalSupply by 10**12. This means that after a few such events, and for a reasonable amount of underlying in the pool `_getExchangeRate() == 0` and all deposits to the pool are blocked.

Agreed that was not clearly stated in the original report. And it technically should not overflow uint.

vnadoda

@clems4ev3r the scenario you described is same as #117, right?

vnadoda

@clems4ev3r can we close this as a duplicate of #117?

clems4ev3r

@vnadoda not exactly the same since this will happen if some funds stay in the contract after locking. Each time a lock happens, depositing back capital will multiply totalSupply by a factor proportional to the locking, eventually forcing _getExchangeRate() to zero and blocking deposits for the ProtectionPool

vnadoda

@clems4ev3r lets' discuss this on the call

vnadoda

@hrishibhat we are planning to fix this

Issue M-7: Freezing of the protocol when totalSTokenUnderlying is zero but totalSupply is non-zero

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/117

Found by

Ruhum, jprod15, Web3SecurityDAO, Kumpa, Bauer, mert_eren, clems4ever, chaduke

Summary

In some cases the protocol can contain zero funds while having a non zero totalSupply of STokens. In that case the protocol will not be able to accept any new deposits and any new protection buys, thus coming to a halt, unless all STokens are burned by their respective holders.

Vulnerability Detail

In the case `lockCapital` has to lock all available capital: https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L415-L419

`totalSTokenUnderlying` becomes zero, but `totalSupply` is still non-zero since no SToken have been burned. Which means that new deposits will revert because `_getExchangeRate()` is zero: https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L602-L605

And `convertToSToken` tries to divide by `_getExchangeRate()`; https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L602-L605

Also all new protection buying attempts will revert because `_leverageRatio` is zero, and thus under `leverageRatioFloor`.

Impact

The protocol comes to a halt, unless every SToken holder burn their shares by calling `withdraw` after enough cycles have passed, returning to the case `totalSupply == 0`.

Code Snippet

Tool used

Manual Review

Recommendation

Keep a minimum amount of totalSTokenUnderlying in the contract in any case (can be 1e6).

Discussion

vnadoda

@clems4ev3r we plan to fix this

Issue M-8: Some protection buyers might not be able to renew their protections due to delayed expiration processing.

Source: https://github.com/sherlock-audit/2023-02-carapace-judging/issues/27

Found by

chaduke

Summary

Some protection buyers will never be able to renew their protection due to delayed expiration processing.

Vulnerability Detail

We show below how some buyers will not be able to renew their protection due to delayed expiration process caused by the the wrong implementation of `verifyAndAccruePremium()`. Due to the bug, they might miss the deadline and grace period for renewal.

1) First the `verifyBuyerCanRenewProtection()` function checks whether it is too late to renew the protection (before the grace period expires). In addition, `verifyBuyerCanRenewProtection()` also checks whether there exists an expired protection with the same lending pool and position token ID at `protectionBuyerAccounts[msg.sender].expiredProtectionIndexByLendingPool[_protectionPurchaseParams.lendingPoolAddress][_protectionPurchaseParams.nftLpTokenId]` in L371. If such an existing expired protection is not there, renewal will be rejected.

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/libraries/ProtectionPoolHelper.sol#L360-L399

2) In order for the expired protection to be stored in `protectionBuyerAccounts[msg.sender].expiredProtectionIndexByLendingPool[_protectionPurchaseParams.lendingPoolAddress][_protectionPurchaseParams.nftLpTokenId]`, the `expireProtection()` must be called at L1004 of function `_accruePremiumAndExpireProtections()`.

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L963-L1021

3) the `expireProtection()` function stores such expired protection at L311-315:

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/libraries/ProtectionPoolHelper.sol#L293-L321

4) However, before `expireProtection()` can be called, function `verifyAndAccruePremium()` needs to be called first to decide whether a protection has expired or not (L990).

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/core/pool/ProtectionPool.sol#L963-L1021

5) However, when there is no payment for a while, for example, when the last payment is made before a protection P starts (`_latestPaymentTimestamp < _startTimestamp`), P will not be considered as `expired` see L215-220 of `verifyAndAccruePremium()` below:

https://github.com/sherlock-audit/2023-02-carapace/blob/main/contracts/libraries/ProtectionPoolHelper.sol#L201-L284

6) Therefore, if there is no payment for a long time, then `verifyAndAccruePremium()` will always consider a protection P has not expired. `expireProtection()` will never process it. So the actually expired protection will not be stored in `protectionBuyerAccounts[msg.sender].expiredProtectionIndexByLendingPool[_protectionPurchaseParams.lendingPoolAddress][_protectionPurchaseParams.nftLpTokenId]`. The buyer for P will not be able to renew P because `verifyBuyerCanRenewProtection()` will fail to find an existing expired protection P. If there is no payment for a long time, then the buyer for P might miss the deadline and grace period and will never be able to renew P anymore.

Impact

Some protection buyers might never be able to renew their protections due to delayed expiration processing as a result of a bug of `verifyAndAccruePremium()`.

`getActiveProtections()` might return some protections that are supposed to have expired, but not processed due to the above bug.

Code Snippet

See above

Tool used

VSCode

Manual Review

Recommendation

We revise `verifyAndAccruePremium()` so that it will process and return the right value when a protection expires.

```
 function verifyAndAccruePremium(
    ProtectionPoolInfo storage poolInfo,
    ProtectionInfo storage protectionInfo,
    uint256 _lastPremiumAccrualTimestamp,
    uint256 _latestPaymentTimestamp
  )
    external
    view
    returns (uint256 _accruedPremiumInUnderlying, bool _protectionExpired)
  {
    uint256 _startTimestamp = protectionInfo.startTimestamp;

+    uint256 _expirationTimestamp = protectionInfo.startTimestamp +
+      protectionInfo.purchaseParams.protectionDurationInSeconds;
+    _protectionExpired = block.timestamp > _expirationTimestamp;


    /// This means no payment has been made after the protection is bought or
 ↪  protection starts in the future.
    /// so no premium needs to be accrued.
-    if (
-      _latestPaymentTimestamp < _startTimestamp ||
-      _startTimestamp > block.timestamp
-    ) {
+    if (!_protectionExpired                                                 //
 ↪  @audit: only if it has not expired
+      (_latestPaymentTimestamp < _startTimestamp ||
+      _startTimestamp > block.timestamp)
+    ) {

      return (0, false);
    }

    /// Calculate the protection expiration timestamp and
    /// Check if the protection is expired or not.
-    uint256 _expirationTimestamp = protectionInfo.startTimestamp +
-      protectionInfo.purchaseParams.protectionDurationInSeconds;
-    _protectionExpired = block.timestamp > _expirationTimestamp;

    /// Only accrue premium if the protection is expired
    /// or latest payment is made after the protection start & last premium
 ↪  accrual
    if (
```

```
      _protectionExpired ||
      (_latestPaymentTimestamp > _startTimestamp &&
        _latestPaymentTimestamp > _lastPremiumAccrualTimestamp)
    ) {
      /**
       * <-Protection Bought(second: 0) --- last accrual ---
↪   now(latestPaymentTimestamp) --- Expiration->
       * The time line starts when protection is bought and ends when protection
↪   is expired.
       * secondsUntilLastPremiumAccrual is the second elapsed since the last
↪   accrual timestamp.
       * secondsUntilLatestPayment is the second elapsed until latest payment is
↪   made.
       */

      // When premium is accrued for the first time, the
↪   _secondsUntilLastPremiumAccrual is 0.
      uint256 _secondsUntilLastPremiumAccrual;
      if (_lastPremiumAccrualTimestamp > _startTimestamp) {
        _secondsUntilLastPremiumAccrual =
          _lastPremiumAccrualTimestamp -
          _startTimestamp;
      }

      /// if loan protection is expired, then accrue premium till expiration and
↪   mark it for removal
      uint256 _secondsUntilLatestPayment;
      if (_protectionExpired) {
        _secondsUntilLatestPayment = _expirationTimestamp - _startTimestamp;
        console.log(
          "Protection expired for amt: %s",
          protectionInfo.purchaseParams.protectionAmount
        );
      } else {
        _secondsUntilLatestPayment = _latestPaymentTimestamp - _startTimestamp;
      }

      /// Calculate the accrued premium amount scaled to 18 decimals
      uint256 _accruedPremiumIn18Decimals = AccruedPremiumCalculator
        .calculateAccruedPremium(
          _secondsUntilLastPremiumAccrual,
          _secondsUntilLatestPayment,
          protectionInfo.K,
          protectionInfo.lambda
        );

      console.log(
        "accruedPremium from second %s to %s: ",
```

```
        _secondsUntilLastPremiumAccrual,
        _secondsUntilLatestPayment,
        _accruedPremiumIn18Decimals
      );

      /// Scale the premium amount to underlying decimals
      _accruedPremiumInUnderlying = scale18DecimalsAmtToUnderlyingDecimals(
        _accruedPremiumIn18Decimals,
        poolInfo.underlyingToken.decimals()
      );
    }
  }
```

Discussion

vnadoda

@clems4ev3r @hrishibhat This will be fixed with daily premium accrual change for #294