SHERLOCK

SHERLOCK SECURITY REVIEW FOR

SHERLOCK

Prepared for:	Sense
Prepared by:	Sherlock
Lead Security Expert:	ak1
Dates Audited:	November 2 - November 9, 2022
Prepared on:	November 22, 2022

Introduction

Sense is decentralized permissionless infrastructure, where teams can build and develop new yield primitives for DeFi.

Scope

The following files in the Sense Auto Roller @ b3f33b repo are in scope.

- `AutoRoller.sol`
- `AutoRollerFactory.sol`
- `RollerPeriphery.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
7	3

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

0x52
ak1
ctf_sec
0xSmartContract
pashov
cryptphi
0xheynacho
8olidity
supernova
minhquanym

Issue H-1: Public vault : Initial depositor can manipulate the price per share value and future depositors are forced to deposit huge value in vault.

Source: https://github.com/sherlock-audit/2022-11-sense-judging/issues/50

Found by

ak1

Summary

Most of the share based vault implementation will face this issue. The vault is based on the ERC4626 where the shares are calculated based on the deposit value. By depositing large amount as initial deposit, initial depositor can influence the future depositors value.

Vulnerability Detail

Shares are minted based on the deposit value. https://github.com/sherlock-audit/2022-11-sense/blob/main/contracts/src/RollerPeriphery.sol#L59-L79 Public vault is based on the ERC4626 where the shares are calculated based on the deposit value.

By depositing large amount as initial deposit, first depositor can take advantage over other depositors.

I am sharing reference for this type of issue that already reported and acknowledged. This explain how the share price could be manipulated to large value.

ERC4626 implementation function mint(uint256 shares, address receiver) public virtual returns (uint256 assets) { assets = previewMint(shares); // No need to check for rounding error, previewMint rounds up.

```
    // Need to transfer before minting or ERC777s could reenter.
    asset.safeTransferFrom(msg.sender, address(this), assets);

    _mint(receiver, shares);

    emit Deposit(msg.sender, receiver, assets, shares);

    afterDeposit(assets, shares);
}

  function previewMint(uint256 shares) public view virtual returns (uint256) {
    uint256 supply = totalSupply; // Saves an extra SLOAD if totalSupply is
    ↪  non-zero.
```

```
    return supply == 0 ? shares : shares.mulDivUp(totalAssets(), supply);
}
```

Impact

Future depositors are forced for huge value of asset to deposit. It is not practically possible for all the users. This could directly affect on the attrition of users towards this system.

Code Snippet

https://github.com/sherlock-audit/2022-11-sense/blob/main/contracts/src/RollerPeriphery.sol#L59-L79

ERC4626 implementation function mint(uint256 shares, address receiver) public virtual returns (uint256 assets) { assets = previewMint(shares); // No need to check for rounding error, previewMint rounds up.

```
    // Need to transfer before minting or ERC777s could reenter.
    asset.safeTransferFrom(msg.sender, address(this), assets);

    _mint(receiver, shares);

    emit Deposit(msg.sender, receiver, assets, shares);

    afterDeposit(assets, shares);
}

  function previewMint(uint256 shares) public view virtual returns (uint256) {
    uint256 supply = totalSupply; // Saves an extra SLOAD if totalSupply is
    ↪  non-zero.

    return supply == 0 ? shares : shares.mulDivUp(totalAssets(), supply);
}
```

Tool used

Manual Review

Recommendation

Consider requiring a minimal amount of share tokens to be minted for the first minter, and send a portion of the initial mints as a reserve to the DAO/ burn so that the price per share can be more resistant to manipulation.

Discussion

Evert0x

Depositor can bypass this initial deposit https://github.com/sherlock-audit/2022-11-sense/blob/main/contracts/src/AutoRoller.sol#L160

Issue H-2: AutoRoller#eject can be used to steal all the yield from vault's YTs

Source: https://github.com/sherlock-audit/2022-11-sense-judging/issues/22

Found by

0x52

Summary

AutoRoller#eject collects all the current yield of the YTs, combines the users share of the PTs and YTs then sends the user the entire target balance of the contract. The problem is that combine claims the yield for ALL YTs, which sends the AutoRoller target assets. Since it sends the user the entire target balance of the contract it accidentally sends the user the yield from all the pool's YTs.

Vulnerability Detail

```
function eject(
    uint256 shares,
    address receiver,
    address owner
) public returns (uint256 assets, uint256 excessBal, bool isExcessPTs) {

    ...

    //@audit call of interest
    (excessBal, isExcessPTs) = _exitAndCombine(shares);

    _burn(owner, shares); // Burn after percent ownership is determined in
    ↪  _exitAndCombine.

    if (isExcessPTs) {
        pt.transfer(receiver, excessBal);
    } else {
        yt.transfer(receiver, excessBal);
    }

    //@audit entire asset (adapter.target) balance transferred to caller, which
    ↪  includes collected YT yield and combined
    asset.transfer(receiver, assets = asset.balanceOf(address(this)));

    emit Ejected(msg.sender, receiver, owner, assets, shares,
        isExcessPTs ? excessBal : 0,
```

```
        isExcessPTs ? 0 : excessBal
    );
}

function _exitAndCombine(uint256 shares) internal returns (uint256, bool) {
    uint256 supply = totalSupply; // Save extra SLOAD.

    uint256 lpBal      = shares.mulDivDown(space.balanceOf(address(this)),
    ↪  supply);
    uint256 totalPTBal = pt.balanceOf(address(this));
    uint256 ptShare    = shares.mulDivDown(totalPTBal, supply);

    ...

    uint256 ytBal = shares.mulDivDown(yt.balanceOf(address(this)), supply);
    ptShare += pt.balanceOf(address(this)) - totalPTBal;

    unchecked {
        // Safety: an inequality check is done before subtraction.
        if (ptShare > ytBal) {

            //@audit call of interest
            divider.combine(address(adapter), maturity, ytBal);
            return (ptShare - ytBal, true);
        } else { // Set excess PTs to false if the balances are exactly equal.
            divider.combine(address(adapter), maturity, ptShare);
            return (ytBal - ptShare, false);
        }
    }
}
```

Eject allows the user to leave the liquidity pool by withdrawing their liquidity from the Balancer pool and combining the PTs and YTs via divider.combine.

https://etherscan.io/address/0x86bA3E96Be68563E41c2f5769F1AF9fAf758e6E0#code#F1#L259

```
function combine(
    address adapter,
    uint256 maturity,
    uint256 uBal
) external nonReentrant whenNotPaused returns (uint256 tBal) {
    if (!adapterMeta[adapter].enabled) revert Errors.InvalidAdapter();
    if (!_exists(adapter, maturity)) revert Errors.SeriesDoesNotExist();

    uint256 level = adapterMeta[adapter].level;
    if (level.combineRestricted() && msg.sender != adapter) revert
    ↪  Errors.CombineRestricted();
```

```
    // Burn the PT
    Token(series[adapter][maturity].pt).burn(msg.sender, uBal);

    //@audit call of interest
    uint256 collected = _collect(msg.sender, adapter, maturity, uBal, uBal,
    ↪  address(0));

    ...

    // Convert from units of Underlying to units of Target
    tBal = uBal.fdiv(cscale);
    ERC20(Adapter(adapter).target()).safeTransferFrom(adapter, msg.sender, tBal);

    // Notify only when Series is not settled as when it is, the _collect() call
    ↪  above would trigger a _redeemYT which will call notify
    if (!settled) Adapter(adapter).notify(msg.sender, tBal, false);
    unchecked {
        // Safety: bounded by the Target's total token supply
        tBal += collected;
    }
    emit Combined(adapter, maturity, tBal, msg.sender);
}
```

https://etherscan.io/address/0x86bA3E96Be68563E41c2f5769F1AF9fAf758e6E0#code#F1#L445

```
function _collect(
    address usr,
    address adapter,
    uint256 maturity,
    uint256 uBal,
    uint256 uBalTransfer,
    address to
) internal returns (uint256 collected) {
    if (!_exists(adapter, maturity)) revert Errors.SeriesDoesNotExist();

    if (!adapterMeta[adapter].enabled && !_settled(adapter, maturity)) revert
    ↪  Errors.InvalidAdapter();

    Series memory _series = series[adapter][maturity];
    uint256 lscale = lscales[adapter][maturity][usr];

    ...

    uint256 tBalNow = uBal.fdivUp(_series.maxscale); // preventive round-up
    ↪  towards the protocol
```

```
    uint256 tBalPrev = uBal.fdiv(lscale);
    unchecked {
        collected = tBalPrev > tBalNow ? tBalPrev - tBalNow : 0;
    }

    //@audit adapter.target is transferred to AutoRoller
    ERC20(Adapter(adapter).target()).safeTransferFrom(adapter, usr, collected);
    Adapter(adapter).notify(usr, collected, false); // Distribute reward tokens

    ...
}
```

Inside divider#combine the collected yield from the YTs are transferred to the AutoRoller. The AutoRoller balance will now contain both the collected yield of the YTs and the target yielded by combining. The end of eject transfers this entire balance to the caller, effectively stealing the yield of the entire AutoRoller.

Impact

User funds given to the wrong person

Code Snippet

https://github.com/sherlock-audit/2022-11-sense/blob/main/contracts/src/AutoRoller.sol#L636-L664

Tool used

Manual Review

Recommendation

Combine returns the amount of target yielded by combining the PT and YT. This balance is the amount of assets that should be transferred to the user.

Discussion

jparklev

This one seems to be valid and we're working on a fix. Appreciate the find

jparklev

Fix: https://github.com/sense-finance/auto-roller/pull/21

aktech297

The fix that is done inside the eject function looks good. Also, there are three new functions are added, I am looking for clarification from @sherlock-admin to confirm whether they are also part of audit and need to be reviewed.

Issue H-3: Adversary can brick AutoRoller by creating another AutoRoller on the same adapter

Source: https://github.com/sherlock-audit/2022-11-sense-judging/issues/20

Found by

0x52

Summary

onSponsorWindowOpened attempts to make a new series at the desired maturity. Each adapter can only have one of each maturity. If the maturity requested already exists then onSponsorWindowOpened will revert, making it impossible to roll the AutoRoller. An adversary can take advantage of this to brick an AutoRoller by creating a second AutoRoller on the same adapter that will create a target maturity before the first AutoRoller. Since the maturity now exists, the first AutoRoller will always revert when trying to Roll.

Vulnerability Detail

```
uint256 _maturity = utils.getFutureMaturity(targetDuration);

function getFutureMaturity(uint256 monthsForward) public view returns (uint256) {
    (uint256 year, uint256 month, ) =
    ↪  DateTime.timestampToDate(DateTime.addMonths(block.timestamp,
    ↪  monthsForward));
    return DateTime.timestampFromDateTime(year, month, 1 /* top of the month */,
    ↪  0, 0, 0);
}
```

Inside AutoRoller#onSponsorWindowOpened the maturity is calculated using RollerUtils#getFutur
This returns the timestamp the requested months ahead, truncated down to the first of the month. It passes this calculated maturity as the maturity to sponsor a new series.

```
(ERC20 _pt, YTLike _yt) = periphery.sponsorSeries(address(adapter), _maturity,
↪  true);
```

https://etherscan.io/address/0xFff11417a58781D3C72083CB45EF54d79Cd02437#code#F1#L90

```
function sponsorSeries(
    address adapter,
    uint256 maturity,
```

```
    bool withPool
) external returns (address pt, address yt) {
    (, address stake, uint256 stakeSize) = Adapter(adapter).getStakeAndTarget();

    // Transfer stakeSize from sponsor into this contract
    ERC20(stake).safeTransferFrom(msg.sender, address(this), stakeSize);

    // Approve divider to withdraw stake assets
    ERC20(stake).approve(address(divider), stakeSize);

    (pt, yt) = divider.initSeries(adapter, maturity, msg.sender);

    // Space pool is always created for verified adapters whilst is optional for
    ↪  unverified ones.
    // Automatically queueing series is only for verified adapters
    if (verified[adapter]) {
        poolManager.queueSeries(adapter, maturity, spaceFactory.create(adapter,
        ↪  maturity));
    } else {
        if (withPool) {
            spaceFactory.create(adapter, maturity);
        }
    }
    emit SeriesSponsored(adapter, maturity, msg.sender);
}
```

periphery#sponsorSeries is called with true indicating to create a space pool for the newly created series.

https://etherscan.io/address/0x5f6e8e9C888760856e22057CBc81dD9e0494aA34#code#F1#L75

```
function create(address adapter, uint256 maturity) external returns (address
↪  pool) {
    address pt = divider.pt(adapter, maturity);
    _require(pt != address(0), Errors.INVALID_SERIES);
    _require(pools[adapter][maturity] == address(0), Errors.POOL_ALREADY_EXISTS);

    pool = address(new Space(
        vault,
        adapter,
        maturity,
        pt,
        ts,
        g1,
        g2,
        oracleEnabled
    ));
```

```
    pools[adapter][maturity] = pool;
}
```

We run into an issue inside SpaceFactory#create because it only allows a single pool per adapter/maturity. If a pool already exist then it will revert.

An adversary can abuse this revert to brick an existing AutoRoller. Assume AutoRoller A has a duration of 3 months. Its current maturity is December 1st 2022, when rolled it will attempt to create a series at March 1st 2023. An adversary could abuse this and create AutoRoller B with a maturity of 4 months. When they roll for the first time it will create a series with maturity at March 1st 2023. When AutoRoller A attempts to roll it will revert since a series already exists at March 1st 2023.

This conflict can happen accidentally if there is a monthly AutoRoller and a quarterly AutoRoller. It also hinders the viability of using an AutoRoller for an adapter that is popular because the series will likely have been created by the time the autoroller tries to roll into it.

Impact

AutoRollers will frequently be bricked

Code Snippet

AutoRoller.sol#L174-L272

Tool used

Manual Review

Recommendation

Requiring that the AutoRoller has to create the series seems overly restrictive and leads to a large number of issues. Attempting to join an a series that is already initialized could also lead to pool manipulation rates. It seems like a large refactoring is needed for the rolling section of the AutoRoller

Discussion

jparklev

This is a great observation and a valid high severity finding. We have a fix in mind that we'll share once implemented – appreciate the submission

jparklev

Fix: https://github.com/sense-finance/auto-roller/pull/20

aktech297

The fix is done by considering the trusted caller. If the caller is trusted, contract allows for auto roller creation. This fix will be valid till `isTrusted[msg.sender]==true` after first one is created. Even the trusted caller should not be allowed to create auto roller with maturity less than the existing auto roller.

Issue M-1: Anyone can spend on behalf of roller periphery

Source: https://github.com/sherlock-audit/2022-11-sense-judging/issues/48

Found by

8olidity, 0x52, supernova, ctf_sec, pashov, cryptphi, minhquanym

Summary

The approve() function in RollerPeriphery contract allows anyone to spend ERC20 token owned by the contract

Vulnerability Detail

RollerPeriphery.approve() does not have any access control, this allows any user to be able to call the approve call which would make an ERC20 approve call to the token inputed, and allowing the 'to' address to spend. In the cases where RollerPeriphery owns some ERC20 tokens. The user will be able to transfer the tokens from the contract as a spender.

Impact

Loss of funds

Code Snippet

https://github.com/sherlock-audit/2022-11-sense/blob/main/contracts/src/RollerPeriphery.sol#L100-L102

```
function approve(ERC20 token, address to, uint256 amount) public payable {
        token.safeApprove(to, amount);
    }
```

ERC20 approve call is:

```
function approve(address spender, uint256 amount) public virtual returns (bool) {
        allowance[msg.sender][spender] = amount;

        emit Approval(msg.sender, spender, amount);

        return true;
    }
```

Tool used

Manual Review

Recommendation

There should be some access control, according to the provided contracts, this function is called by RollerFactory, this can be the only address allowed to call the RollerPeriphery.approve() function.

Discussion

jparklev

We don't expect that the Periphery will ever hold onto funds of its own, so this is acceptable behavior to us. However, the DOS version of this ticket #46 might be valid as a `medium`

Evert0x

Grouping all as medium as they point out the same flaw with different impacts.

Issue M-2: Missing ReEntrancy Guard to `claimRewards` function

Source: https://github.com/sherlock-audit/2022-11-sense-judging/issues/40

Found by

0xSmartContract

Summary

There is no re-entry risk on true ERC-20 tokens that work according to the spec (i.e. audited, etc.).

However you can write a malicious ERC-20 with custom safetransferFrom() or approve() that have re-entrancy hooks to attack a target.

Furthermore ERC-777 is backwards compatible token standard with ERC-20 standard. ERC-777 has better usability, but it has transfer hooks that can cause reentrancy.

Vulnerability Detail

ERC20 generally doesn't result in reentrancy, however ERC777 tokens can and they can maskerade as ERC20. So if a contract interacts with unknown ERC20 tokens it is better to be safe and consider that transfers can create reentrancy problems.

Impact

Although reentrancy attack is considered quite old over the past two years, there have been cases such as:

Uniswap/LendfMe hacks (2020) ($25 mln, attacked by a hacker using a reentrancy)

The BurgerSwap hack (May 2021) ($7.2 million because of a fake token contract and a reentrancy exploit.)

The SURGEBNB hack (August 2021) ($4 million seems to be a reentrancy-based price manipulation attack.)

CREAM FINANCE hack (August 2021) ($18.8 million, reentrancy vulnerability allowed the exploiter for the second borrow.)

Siren protocol hack (September 2021) ($3.5 million, AMM pools were exploited through reentrancy attack.)

Type of Reentrancy

Details 1 - Single Function Reentrancy 2 - Cross-Function Reentrancy 3 - Cross-Contract Reentrancy

Code Snippet

Must be re-entrancy guard to below functions;

AutoRoller.sol#L654-L659

AutoRoller.sol#L715

```
contracts/src/AutoRoller.sol:
  709       /// @param coin address of the coin to transfer out.
  710:      function claimRewards(ERC20 coin) external {
  711:          require(coin != asset);
  712:          if (maturity != MATURITY_NOT_SET) {
  713:              require(coin != ERC20(address(yt)) && coin != pt && coin !=
↪  ERC20(address(space)));
  714:          }
  715:          coin.transfer(rewardRecipient, coin.balanceOf(address(this)));
  716:      }


contracts/src/AutoRoller.sol:
  635       /// @return isExcessPTs Whether the excess token is a YT or PT.
  636:      function eject(
  637:          uint256 shares,
  638:          address receiver,
  639:          address owner
  640:      ) public returns (uint256 assets, uint256 excessBal, bool isExcessPTs)
↪  {
  641:          if (maturity == MATURITY_NOT_SET) revert ActivePhaseOnly();
  642:
  643:          if (msg.sender != owner) {
  644:              uint256 allowed = allowance[owner][msg.sender]; // Saves gas
↪  for limited approvals.
  645:
  646:              if (allowed != type(uint256).max) allowance[owner][msg.sender]
↪  = allowed - shares;
  647:          }
  648:
  649:          (excessBal, isExcessPTs) = _exitAndCombine(shares);
  650:
  651:          _burn(owner, shares); // Burn after percent ownership is
↪  determined in _exitAndCombine.
  652:
  653:          if (isExcessPTs) {
  654:              pt.transfer(receiver, excessBal);
```

```
655:         } else {
656:             yt.transfer(receiver, excessBal);
657:         }
658:
659:         asset.transfer(receiver, assets = asset.balanceOf(address(this)));
660:         emit Ejected(msg.sender, receiver, owner, assets, shares,
661:             isExcessPTs ? excessBal : 0,
662:             isExcessPTs ? 0 : excessBal
663:         );
664:     }
```

Tool used

Manual Review

Recommendation

Use Openzeppelin or Solmate Re-Entrancy pattern

Here is a example of a re-entracy guard

```
pragma solidity 0.8.13;

contract ReEntrancyGuard {
    bool internal locked;

    modifier noReentrant() {
        require(!locked, "No re-entrancy");
        locked = true;
        _;
        locked = false;
    }
}
```

Discussion

jparklev

Assets are not expected to re-enter and we don't think reward tokens re-entering on claimRewards would cause a problem. Nevertheless, it's a fair suggestion overall and we'll likely make this change

jparklev

Fix: https://github.com/sense-finance/auto-roller/pull/19

aktech297

Changes are fine. I would suggest to add the reentrancy guard for eject function too.

Issue M-3: Vulnerability related to 'Optimizer Bug Regarding Memory Side Effects of Inline Assembly'

Source: https://github.com/sherlock-audit/2022-11-sense-judging/issues/39

Found by

0xSmartContract, 0xheynacho

Summary

Solidity versions 0.8.13 and 0.8.14 are vulnerable to a recently reported optimizer bug related to inline assembly. Solidity 0.8.15 has been released with a fix.

Vulnerability Detail

AutoRoller.sol inherits solidity contracts from solmate FixedPointMathLib and ERC4626, and both these uses inline assembly, and optimization is enabled while compiling.

Impact

This bug only occurs under very specific conditions: the legacy optimizer must be enabled rather than the IR pipeline (true for the current project configuration), and the affected assembly blocks must not refer to any local Solidity variables.

Code Snippet

AutoRoller.sol#L5

AutoRoller.sol#L7

FixedPointMathLib.sol#L36-L51

```
contracts/src/AutoRoller.sol:

  4: import { ERC20 } from "solmate/tokens/ERC20.sol";
  5: import { FixedPointMathLib } from "solmate/utils/FixedPointMathLib.sol";
  6: import { SafeTransferLib } from "solmate/utils/SafeTransferLib.sol";
  7: import { ERC4626 } from "solmate/mixins/ERC4626.sol";

contracts/foundry.toml:
  1: [profile.default]
  8: optimizer_runs = 10000
```

Tool used

Manual Review

Recommendation

Use recent Solidity version 0.8.15 which has the fix for these issues.

Discussion

jparklev

We will bump our solidity patch version

jparklev

Fix: https://github.com/sense-finance/auto-roller/pull/18

aktech297

Verified the fix. The contracts are updated with 0.8.15 compiler version.

Issue M-4: AutoRoller.sol#roll can revert if lastSettle is zero because solmate ERC4626 deposit revert if previewDeposit returns 0

Source: https://github.com/sherlock-audit/2022-11-sense-judging/issues/33

Found by

ctf_sec

Summary

AutoRoller.sol#roll can revert if lastSettle is zero because solmate ERC4626 deposit revert if previewDeposit returns 0

Vulnerability Detail

let us look into the implementation of function roll()

```
/// @notice Roll into the next Series if there isn't an active series and the
 ↪  cooldown period has elapsed.
function roll() external {
    if (maturity != MATURITY_NOT_SET) revert RollWindowNotOpen();

    if (lastSettle == 0) {
        // If this is the first roll, lock some shares in by minting them for the
 ↪  zero address.
        // This prevents the contract from reaching an empty state during future
 ↪  active periods.
        deposit(firstDeposit, address(0));
    } else if (lastSettle + cooldown > block.timestamp) {
        revert RollWindowNotOpen();
    }

    lastRoller = msg.sender;
    adapter.openSponsorWindow();
}
```

note, if lastSettle is 0, we deposit a small amount of token and mint shares to address(0)

```
deposit(firstDeposit, address(0));
```

First deposit is a fairly small amount:

```
firstDeposit  = (0.01e18 - 1) / scalingFactor + 1;
```

We can deposit from ERC4626 implementation:

```
function deposit(uint256 assets, address receiver) public virtual returns
↪  (uint256 shares) {
    // Check for rounding error since we round down in previewDeposit.
    require((shares = previewDeposit(assets)) != 0, "ZERO_SHARES");

    // Need to transfer before minting or ERC777s could reenter.
    asset.safeTransferFrom(msg.sender, address(this), assets);

    _mint(receiver, shares);

    emit Deposit(msg.sender, receiver, assets, shares);

    afterDeposit(assets, shares);
}
```

note the restriction:

```
// Check for rounding error since we round down in previewDeposit.
require((shares = previewDeposit(assets)) != 0, "ZERO_SHARES");

// Need to transfer before minting or ERC777s could reenter.
asset.safeTransferFrom(msg.sender, address(this), assets);
```

if previewDeposit returns 0 shares, transaction revert. Can previewDeposit returns 0 shares? it is very possible.

```
function previewDeposit(uint256 assets) public view override returns (uint256) {
    if (maturity == MATURITY_NOT_SET) {
        return super.previewDeposit(assets);
    } else {
        Space _space = space;
        (uint256 ptReserves, uint256 targetReserves) = _getSpaceReserves();

        // Calculate how much Target we'll end up joining the pool with, and use
↪   that to preview minted LP shares.
        uint256 previewedLPBal = (assets - _getTargetForIssuance(ptReserves,
↪   targetReserves, assets, adapter.scaleStored()))
            .mulDivDown(_space.adjustedTotalSupply(), targetReserves);

        // Shares represent proportional ownership of LP shares the vault holds.
        return previewedLPBal.mulDivDown(totalSupply,
↪   _space.balanceOf(address(this)));
```

```
    }
}
```

If (previewedLPBal * total) / space balance is truncated to 0, transaction revert. _space.balanceOf can certainly be inflated if malicious actor send the space token to the address manually. Or previewedLPBal * total could just be small and the division is truncated to 0.

Impact

calling roll would revert and the new sponsored series cannot be started properly.

Code Snippet

https://github.com/sherlock-audit/2022-11-sense/blob/main/contracts/src/AutoRoller.sol#L152-L168

https://github.com/sherlock-audit/2022-11-sense/blob/main/contracts/src/AutoRoller.sol#L416-L435

Tool used

Manual Review

Recommendation

We recommend the project not deposit a such small amount, or there could be a function that let admin gradually control how many tokens should we put in the first deposit.

Discussion

jparklev

While this is an interesting observation and it got us thinking, we don't think that it's something to be concerned about for two reasons:

1) roll would be unaffected since the deposit before roll happens during a cooldown period, where the `previewedLPBal.mulDivDown(totalSupply,_space.balanceOf(address(this)))` line is never executed
2) `previewedLPBal` is a function of total supply, so if `_space.balanceOf(address(this))` were made larger, so to would `previewedLPBal`

We aren't yet on 100% confidence here tho, so if there were to be a test case demonstrating a concrete example of how this could happen, it would be much appreciated

aktech297

I believe if this needs to happen, user have to waste a large amount of fund to do it for no personal gain. I see the #41 has fix for this. Further adding, a simple test is needed to ensure the function flow to confirm when the else part would be executed. I don't see the actual flow from the explanation.

```
 else {
    Space _space = space;
    (uint256 ptReserves, uint256 targetReserves) = _getSpaceReserves();

    // Calculate how much Target we'll end up joining the pool with, and use that
    ↪  to preview minted LP shares.
    uint256 previewedLPBal = (assets - _getTargetForIssuance(ptReserves,
    ↪  targetReserves, assets, adapter.scaleStored()))
        .mulDivDown(_space.adjustedTotalSupply(), targetReserves);

    // Shares represent proportional ownership of LP shares the vault holds.
    return previewedLPBal.mulDivDown(totalSupply,
    ↪  _space.balanceOf(address(this)));
}
```

Issue M-5: Math rounding in AutoRoller.sol is not ERC4626-complicant: previewWithdraw should round up.

Source: https://github.com/sherlock-audit/2022-11-sense-judging/issues/30

Found by

ctf_sec

Summary

Math rounding in AutoRoller.sol is not ERC4626-complicant: previewWithdraw should round up.

Vulnerability Detail

Per EIP 4626's Security Considerations (https://eips.ethereum.org/EIPS/eip-4626)

> Finally, ERC-4626 Vault implementers should be aware of the need for specific, opposing rounding directions across the different mutable and view methods, as it is considered most secure to favor the Vault itself during calculations over its users:
>
> If (1) it's calculating how many shares to issue to a user for a certain amount of the underlying tokens they provide or (2) it's determining the amount of the underlying tokens to transfer to them for returning a certain amount of shares, it should round down. If (1) it's calculating the amount of shares a user has to supply to receive a given amount of the underlying tokens or (2) it's calculating the amount of underlying tokens a user has to provide to receive a certain amount of shares, it should round up.

Then previewWithdraw in AutoRoller.sol should round up.

The original implementation for previewWithdraw in Solmate ERC4626 is:

```
function previewWithdraw(uint256 assets) public view virtual returns (uint256) {
    uint256 supply = totalSupply; // Saves an extra SLOAD if totalSupply is
↪   non-zero.

    return supply == 0 ? assets : assets.mulDivUp(supply, totalAssets());
}
```

It is rounding up, however in the implementation of the AutoRoller.sol#previewWith is not round up.

```
for (uint256 i = 0; i < 20;) { // 20 chosen as a safe bound for convergence from
↪   practical trials.
```

```
    if (guess > supply) {
        guess = supply;
    }

    int256 answer = previewRedeem(guess.safeCastToUint()).safeCastToInt() -
↪   assets.safeCastToInt();

    if (answer >= 0 && answer <= assets.mulWadDown(0.001e18).safeCastToInt() ||
↪   (prevAnswer == answer)) { // Err on the side of overestimating shares needed.
↪   Could reduce precision for gas efficiency.
        break;
    }

    if (guess == supply && answer < 0) revert InsufficientLiquidity();

    int256 nextGuess = guess - (answer * (guess - prevGuess) / (answer -
↪   prevAnswer));
    prevGuess  = guess;
    prevAnswer = answer;
    guess      = nextGuess;

    unchecked { ++i; }
}

return guess.safeCastToUint() + maxError; // Buffer for pow discrepancies.
```

note the line:

```
int256 answer = previewRedeem(guess.safeCastToUint()).safeCastToInt() -
↪   assets.safeCastToInt();
```

previewRedeem is round down.

and later we update guess and return guess

```
int256 nextGuess = guess - (answer * (guess - prevGuess) / (answer -
↪   prevAnswer));
prevGuess  = guess;
prevAnswer = answer;
guess      = nextGuess;
```

and

```
return guess.safeCastToUint() + maxError; // Buffer for pow discrepancies.
```

when calculating the the nextGuess, the code does not round up.

```
int256 nextGuess = guess - (answer * (guess - prevGuess) / (answer -
↪  prevAnswer));
```

Impact

Other protocols that integrate with Sense finance AutoRoller.sol might wrongly assume that the functions handle rounding as per ERC4626 expectation. Thus, it might cause some intergration problem in the future that can lead to wide range of issues for both parties.

Code Snippet

https://github.com/sherlock-audit/2022-11-sense/blob/main/contracts/src/AutoRoller.sol#L528-L567

Tool used

Manual Review

Recommendation

Round up in previewWithdraw using mulDivUp and divWadUp

Discussion

jparklev

Our understanding is that `nextGuess` does not need to be rounded up since it's just a "guess" that is confirmed or denied in how close the "answer" is to what we're looking for. So the rounding needs to be in the answer assessment stage.

In addition, as our comment in the answer inequality says, we do overestimate the shares needed, which is equivalent to rounding up. Perhaps one could make the case that the inequality should be `>0` rather than `>=0` so that exact matches from the rounded down `previewRedeem` don't make it through

Given the above, we're ok accepting this issue, but disagree with the severity

jparklev

Fix: https://github.com/sense-finance/auto-roller/pull/22

Evert0x

@jparklev What severity are you suggesting?

jparklev

> @jparklev What severity are you suggesting?

`medium` would be our suggestion

aktech297

Verified the fix. As @jparklev mentioned, `roundingneedstobeintheanswerassessment stage`, the fix is not related to rounding up. It is related to inequality. so, the fix is to check for `>0` rather than `>=0` so that exact matches from the rounded down `preview Redeem` don't make it through.

Issue M-6: Hardcoded divider address in RollerUtils is incorrect and will brick autoroller

Source: https://github.com/sherlock-audit/2022-11-sense-judging/issues/19

Found by

0x52

Summary

RollerUtils uses a hard-coded constant for the Divider. This address is incorrect and will cause a revert when trying to call AutoRoller#cooldown. If the adapter is combineRestricted then LPs could potentially be unable to withdraw or eject.

Vulnerability Detail

```
address internal constant DIVIDER = 0x09B10E45A912BcD4E80a8A3119f0cfCcad1e1f12;
```

RollerUtils uses a hardcoded constant DIVIDER to store the Divider address. There are two issues with this. The most pertinent issue is that the current address used is not the correct mainnet address. The second is that if the divider is upgraded, changing the address of the RollerUtils may be forgotten.

```
(, uint48 prevIssuance, , , , , uint256 iscale, uint256 mscale, ) =
↪  DividerLike(DIVIDER).series(adapter, prevMaturity);
```

With an incorrect address the divider#series call will revert causing RollerUtils#getNewTargetedRa to revert, which is called in AutoRoller#cooldown. The result is that the AutoRoller cycle can never be completed. LP will be forced to either withdraw or eject to remove their liquidity. Withdraw only works to a certain point because the AutoRoller tries to keep the target ratio. After which the eject would be the only way for LPs to withdraw. During eject the AutoRoller attempts to combine the PT and YT. If the adapter is also combineRestricted then there is no longer any way for the LPs to withdraw, causing loss of their funds.

Impact

Incorrect hard-coded divider address will brick autorollers for all adapters and will cause loss of funds for combineRestricted adapters

Code Snippet

AutoRoller.sol#L853-L914

Tool used

Manual Review

Recommendation

RollerUtils DIVIDER should be set by constructor. Additionally RollerUtils should be deployed by the factory constructor to make sure they always have the same immutable divider reference.

Discussion

jparklev

> RollerUtils uses a hardcoded constant DIVIDER to store the Divider address. There are two issues with this. The most pertinent issue is that the current address used is not the correct mainnet address. The second is that if the divider is upgraded, changing the address of the RollerUtils may be forgotten.

We consider upgrading the divider a very disruptive and unlikely change that we'd need to redeploy most of our system for anyway, so we're not concerned with hardcoding as is. In addition, the address is for the goerli divider, so it was valid in the context we were testing it in.

That said, as we thought about it again, the divider address should actually be an immutable passed in via constructor, so we're validating this one and appreciate having a reason to rethink our assumptions

jparklev

Fix: https://github.com/sense-finance/auto-roller/pull/17

aktech297

Changes looks fine. As a suggestion, input address can be validated.

Issue M-7: Code does not handle ERC20 tokens with special `transfer` implementation

Source: https://github.com/sherlock-audit/2022-11-sense-judging/issues/10

Found by

cryptphi, pashov

Summary

Calls to ERC20::transfer method should always be checked

Vulnerability Detail

Some ERC20 tokens do not revert on failure in `transfer` but instead return `false` as a return value (for example ZRX). Because of this it has become a common practice to use OpenZeppelin's SafeERC20 to handle such weird tokens. If `transfer` fails, but does not revert it can leave tokens stuck in the contract - for example in `eject` in `AutoRoller` we have such a non-checked `transfer`, but if it failed the tokens would get stuck, before the shares used for `eject` were already burned.

Impact

The impact is potentially permanently lost (stuck) value for users of the protocol, but it needs a special ERC20 token to be used as `underlying` or to be sent in contract by mistake, hence Medium severity.

Code Snippet

https://github.com/sherlock-audit/2022-11-sense/blob/main/contracts/src/AutoRoller.sol#L656 https://github.com/sherlock-audit/2022-11-sense/blob/main/contracts/src/AutoRoller.sol#L659 https://github.com/sherlock-audit/2022-11-sense/blob/main/contracts/src/AutoRoller.sol#L715

Tool used

Manual Review

Recommendation

Use OpenZeppelin's SafeERC20 library to handle such tokens

Discussion

jparklev

We will add the safe transfer functions to the remaining locations

jparklev

Fix: https://github.com/sense-finance/auto-roller/pull/16

aktech297

The fix is using `safeTransfer` from `solmate/utils/SafeTransferLib.sol` (asset, coin)
As a suggestion, following `transfer` also can be updated with `safeTransfer` Line - 654,656.