hexens

HATS FINANCE SMART CONTRACT AUDIT REPORT

09.12.2022

CONTENTS

SUMMARY

SEVERITY	NUMBER OF FINDINGS
CRITICAL	0
HIGH	0
MEDIUM	1
LOW	3
INFORMATIONAL	2

TOTAL: 6

SCOPE

The analyzed resources are located on:
https://github.com/hats-finance/hats-contracts/commit/77761c4f6197872ab9b7fc7ec6e43fc14193fbab

The issues described in the report were fixed in the following commit:
https://github.com/hats-finance/hats-contracts/commit/329a783fe12c4b37ff56bd456f2aa450fd80d107

WEAKNESSES

This section contains the list of discovered weaknesses.

1. SLIPPAGE PROTECTION MISSING FOR THE VAULT

SEVERITY: Medium

PATH: HATVault.sol:withdraw, redeem, deposit, mint

REMEDIATION: add a parameter that allows the user to define the amount of maximal/minimal shares and assets for deposit/withdraw/redeem/mint calls respectively. These values can then be used to check for the slippage and protect the user from unexpected changes, as well as implement the appropriate functionality in the front-end for EOAs. Please refer to EIP-5143 (https://eips.ethereum.org/EIPS/eip-5143) for further details

STATUS: fixed

DESCRIPTION:

HATVault implements **ERC4626** for vault mechanics, however the **OpenZeppelin** implementation does not have slippage protection by default. The deriving contract should implement slippage checks if it allows **EOAs** to interact (as stated in EIP-5143), which is the case for **HATVault**.

As a result, the bug reporter can arrange a bounty claim call by frontrunning a vault withdrawal or backrunning a deposit call to maximise profit.

```
function withdraw(uint256 assets, address receiver, address owner)
  public override(IHATVault, ERC4626Upgradeable) virtual returns (uint256) {
  (uint256 _shares, uint256 _fee) = previewWithdrawAndFee(assets);
  _withdraw(_msgSender(), receiver, owner, assets, _shares, _fee);
  return _shares;
}

function redeem(uint256 shares, address receiver, address owner)
  public override(IHATVault, ERC4626Upgradeable) virtual returns (uint256) {
  (uint256 _assets, uint256 _fee) = previewRedeemAndFee(shares);
  _withdraw(_msgSender(), receiver, owner, _assets, shares, _fee);

  return _assets;
}

function deposit(uint256 assets, address receiver) public override(IHATVault, ERC4626Upgradeable)
virtual returns (uint256) {
  return super.deposit(assets, receiver);
}
```

2. USING CONTEXT MIXIN AND MSG.SENDER

SEVERITY: Low

PATH: HATVault.sol:withdrawRequest:L499-505

REMEDIATION: consider using _msgSender() mixin in the withdrawRequest function

STATUS: fixed

DESCRIPTION:

In **HATVault.sol** the functions that overload **ERC4626** methods use the **_msgSender()** context mixin in case of a forwarder or meta-transactions will be implemented in the future. The function **withdrawRequest** is accessing **msg.sender** directly in the code, although it is part of the vault's deposit/withdraw mechanics; this situation can potentially bring major discrepancies in accounting and the overall logic of the contract in the future.

```
function withdrawRequest() external nonReentrant {
  // set the withdrawEnableStartTime time to be withdrawRequestPendingPeriod from now
  // solhint-disable-next-line not-rely-on-time
  uint256 _withdrawEnableStartTime = block.timestamp +
registry.getWithdrawRequestPendingPeriod();
  withdrawEnableStartTime[msg.sender] = _withdrawEnableStartTime;
  emit WithdrawRequest(msg.sender, _withdrawEnableStartTime);
}

function emergencyWithdraw(address receiver) external returns (uint256 assets) {
  _isEmergencyWithdraw = true;
  assets = redeem(balanceOf(_msgSender()), receiver, _msgSender());
  _isEmergencyWithdraw = false;
}
 function withdraw(uint256 assets, address receiver, address owner)
  public override(IHATVault, ERC4626Upgradeable) virtual returns (uint256) {
  (uint256 _shares, uint256 _fee) = previewWithdrawAndFee(assets);
  _withdraw(_msgSender(), receiver, owner, assets, _shares, _fee);

  return _shares;
}

function redeem(uint256 shares, address receiver, address owner)
  public override(IHATVault, ERC4626Upgradeable) virtual returns (uint256) {
  (uint256 _assets, uint256 _fee) = previewRedeemAndFee(shares);
  _withdraw(_msgSender(), receiver, owner, _assets, shares, _fee);

  return _assets;
}
```

3. DUPLICATE BENEFICIARIES WOULD MULTIPLY BOUNTY PAYOUT

SEVERITY: Low

PATH: HATVaultsRegistry.sol:swapAndSend:L335-392

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

In **HATVaultsRegistry.sol:swapAndSend** (L335-392) the parameter **_beneficiaries** is a list of addresses of users that still require their bounties to be swapped to HAT and paid out. However, the function first loops over all addresses and keeps track of each payout per index. The mapping containing the reward for the user is only updated after the payout. If the list would contain a duplicate, the user would receive their bounty multiple times.

```
function swapAndSend(
  address _asset,
  address[] calldata _beneficiaries,
  uint256 _amountOutMinimum,
  address _routingContract,
  bytes calldata _routingPayload
) external onlyOwner {
  SwapData memory _swapData;
  _swapData.hackerRewards = new uint256[](_beneficiaries.length);
  _swapData.governanceHatReward = governanceHatReward[_asset];
  _swapData.amount = _swapData.governanceHatReward;
  for (uint256 i = 0; i < _beneficiaries.length;) {
     _swapData.hackerRewards[i] = hackersHatReward[_asset][_beneficiaries[i]];
     _swapData.amount += _swapData.hackerRewards[i];
     unchecked { ++i; }
  }
  [..]
}
```

We recommend to check the **_beneficiaries** parameter for any duplicates. This can be implemented efficiently in the same loop that extracts the rewards, maintaining the complexity O(n), as follows:

```
address _lastHacker;
for (uint256 i = 0; i < _beneficiaries.length;) {
  require(_beneficiaries[i] > _lastHacker, "Duplicate or not sorted");
  _swapData.hackerRewards[i] = hackersHatReward[_asset][_beneficiaries[i]];
  _swapData.amount += _swapData.hackerRewards[i];
  unchecked { ++i; }
}
```

4. CENTRALIZATION RISK

SEVERITY: Low

PATH: HATVaultsRegistry.sol:_swapTokenForHAT:L451-479

REMEDIATION: we would recommend to either have _swapTokenForHAT call into the right controller contract that performs the swap (e.g. a UniswapV3 controller that finds the right pool) or to validate that the amount of HAT received is roughly equal to assets used according to some price oracle, this way you can validate that the value of assets in the contract stayed roughly the same after the external

STATUS: acknowledged

DESCRIPTION:

The function **HATVaultsRegistry.sol:_swapTokenForHAT** is called when the owner of the registry calls **swapAndSend**. All of the assets that are awarded as bounties for hackers will be approved to a contract address and subsequently an external call is made to this contract with arbitrary call data, both values are given as parameters.

The function does not validate that the funds will actually be used for swapping and as a result poses a centralization risk for rewards if the owner gets compromised.

```
function _swapTokenForHAT(
  IERC20 _asset,
  uint256 _amount,
  uint256 _amountOutMinimum,
  address _routingContract,
  bytes calldata _routingPayload)
internal
returns (uint256 hatsReceived, uint256 amountUnused)
{
  IERC20 _HAT = HAT;
  if (_asset == _HAT) {
     return (_amount, 0);
  }

  IERC20(_asset).safeApprove(_routingContract, _amount);
  uint256 _balanceBefore = _HAT.balanceOf(address(this));
  uint256 _assetBalanceBefore = _asset.balanceOf(address(this));

  // solhint-disable-next-line avoid-low-level-calls
  (bool success,) = _routingContract.call(_routingPayload);
  if (!success) revert SwapFailed();
  hatsReceived = _HAT.balanceOf(address(this)) - _balanceBefore;
  amountUnused = _amount - (_assetBalanceBefore - _asset.balanceOf(address(this)));
  if (hatsReceived < _amountOutMinimum)
     revert AmountSwappedLessThanMinimum();

  IERC20(_asset).safeApprove(address(_routingContract), 0);
}
```

5. REDUNDANT RETURN STATEMENT

SEVERITY: Informational

PATH: HATToken.sol:burn:L32-35

REMEDIATION: delete the return statement and replace with only the call to _burn

STATUS: fixed

DESCRIPTION:

In the function **HATToken.sol:burn**, the **return** statement has no effect.

```
function burn(uint256 _amount) external {
  if (_amount == 0) revert ZeroAmount();
  return _burn(msg.sender, _amount);
}
```

6. INCORRECT COMPARISON FOR TIMESTAMP

SEVERITY: Informational

PATH: PATH: HATVault.sol:approveClaim:L273

REMEDIATION: HATVault.sol:challengeClaim the comparison should be replaced by

block.timestamp >= activeClaim.createdAt + activeClaim.challengePeriod

so that activeClaim.challengedAt will be strictly smaller than activeClaim.createdAt + activeClaim.challengePeriod.

STATUS: fixed

DESCRIPTION:

In **HATVault.sol:challengeClaim** (L243-255) a claim can be challenged, unless block.timestamp > activeClaim.createdAt + activeClaim.challengePeriod is true. This means that activeClaim.challengeAt can be exactly equal to activeClaim.createdAt + activeClaim.challengePeriod.

In **HATVault.sol:approveClaim** (L258-287) a claim can be approved, unless it is expired. A claim is considered expired in both approveClaim and dismissClaim if block.timestamp >= _claim.createdAt + _claim.challengePeriod + _claim.challengeTimeOutPeriod is true.

However, in **approveClaim** on L273 the check to see if the claim is still in the challenge period (revert if block.timestamp > _claim.challengedAt + _claim.challengeTimeOutPeriod) includes the case where block.timestamp == _claim.createdAt + _claim.challengePeriod + _claim.challengeTimeOutPeriod. This case occurs if _claim.challengedAt == _claim.createdAt + _claim.challengePeriod, which is possible in challengeClaim, but it will never be reached because the claim would have expired.

```
function challengeClaim(bytes32 _claimId) external isActiveClaim(_claimId) {
  if (msg.sender != activeClaim.arbitrator && msg.sender != registry.owner())
     revert OnlyArbitratorOrRegistryOwner();
  if (block.timestamp > activeClaim.createdAt + activeClaim.challengePeriod)
     revert ChallengePeriodEnded();
  if (activeClaim.challengedAt != 0) {
     revert ClaimAlreadyChallenged();
  }
  activeClaim.challengedAt = uint32(block.timestamp);
  emit ChallengeClaim(_claimId);
}
function approveClaim(bytes32 _claimId, uint16 _bountyPercentage) external nonReentrant isActiveClaim(_claimId) {
  Claim memory _claim = activeClaim;
  delete activeClaim;
  if (block.timestamp >= _claim.createdAt + _claim.challengePeriod + _claim.challengeTimeOutPeriod) {
     revert ClaimExpired();
  }
  if (_claim.challengedAt != 0) {       if (
        msg.sender != _claim.arbitrator ||
        block.timestamp > _claim.challengedAt + _claim.challengeTimeOutPeriod
     )
        revert ChallengedClaimCanOnlyBeApprovedByArbitratorUntilChallengeTimeoutPeriod();
     // the arbitrator can update the bounty if needed
     if (_claim.arbitratorCanChangeBounty && _bountyPercentage != 0) {
        _claim.bountyPercentage = _bountyPercentage;
     }
  } else {
     if (
        block.timestamp <= _claim.createdAt + _claim.challengePeriod
     )
        revert UnchallengedClaimCanOnlyBeApprovedAfterChallengePeriod();
  }
  [..]
}
function dismissClaim(bytes32 _claimId) external isActiveClaim(_claimId) {
  uint256 _challengeTimeOutPeriod = activeClaim.challengeTimeOutPeriod;
  if (block.timestamp < activeClaim.createdAt + activeClaim.challengePeriod + _challengeTimeOutPeriod) {
     [..]
  } // else the claim is expired and should be dismissed
  [..]
}
```

hexens