SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Public
Prepared for:	Exactly Protocol
Prepared by:	Sherlock
Lead Security Expert:	0x73696d616f
Dates Audited:	July 22 - July 25, 2024
Prepared on:	August 26, 2024

Introduction

The new Exactly staking program will enable EXA token holders to earn dividends and gain additional voting power by staking their EXA tokens. This structured system of rewards will be based on the staking duration and the amount staked, promoting long-term commitment and contributing to the growth and stability of the Exactly Protocol.

Scope

Repository: exactly/protocol

Branch: next

Commit: 0872c11bc7fc6bcdbe630f072c688a448beacbe0

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

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

Issues found

Medium	High
13	0

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

0x73696d616f
KupiaSec
santipu_
sakshamguruji
HackTrace

0xBugHunter
blockchain555
cawfree
eeyore
Bigsam

HaxSecurity
santiellena
0uts1der

Issue M-1: Depositing to another receiver othan than `msg.sender` will lead to stuck funds by increasing `avgStart` without claiming

Source: https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts-judging/issues/21

Found by

0x73696d616f, 0xBugHunter, blockchain555, cawfree

Summary

`StakedEXA::_update()` is called when minting and if the time staked is smaller than the reference time, it calls `_claim(reward), which claims the rewards for the` msg.sender`. However, the mint may be to someone else other than` msg.sender`, so` avgStart[to]' will increase without first claiming, which will lead to loss of rewards for the receiver of the new deposit.

Root Cause

In `StakedEXA.sol::146` on claim(reward), it claims to the `msg.sender`, but the deposit is done to the `to` address, which may be different.

Internal pre-conditions

1. Anyone must do a deposit to a different address.

External pre-conditions

None.

Attack Path

1. User has some StakedEXA from previous deposits.
2. Another user deposits/mints to user 1, which increases `avgStart[user1]`, but does not claim first for user1, making the funds stuck.

Impact

The funds are stuck for much longer than they are supposed to. The duration is `24 weeks` in the tests, so users can have to wait up to 24 weeks more if they do a really

big deposit while having fully claimable rewards. Additionally, in case they decide to withdraw after depositing, before the rewards vest again, they will lose these funds to savings.

PoC

Paste the following test in `StakedEXA.t.sol` to confirm that `address(this)` deposits more without claiming.

```
function test_POC_increaseBalance_withoutClaim() external {
  uint256 assets = 1e18;
  uint256 time = minTime;

  exa.mint(address(this), assets);
  stEXA.deposit(assets, address(this));

  uint256 initTime = block.timestamp * 1e18;
  skip(time + 1);
  uint256 finalTime = block.timestamp * 1e18;

  assertEq(rA.balanceOf(address(this)), 0);
  assertEq(stEXA.avgStart(address(this)), initTime);
  assertEq(stEXA.rawClaimable(rA, address(this), assets), 20833367779982251207);
  assertEq(stEXA.earned(rA, address(this), assets), 41666735559964287164);

  address anotherAccount = makeAddr("AnotherAccount");
  exa.mint(anotherAccount, assets);
  vm.startPrank(anotherAccount);
  exa.approve(address(stEXA), assets);
  stEXA.deposit(assets, address(this));
  vm.stopPrank();

  uint256 newAssets = 2*assets;

  assertEq(rA.balanceOf(address(this)), 0);
  assertEq(stEXA.avgStart(address(this)), (initTime + finalTime) / 2);
  assertEq(stEXA.rawClaimable(rA, address(this), newAssets), 0);
  assertEq(stEXA.earned(rA, address(this), newAssets), 41666735559964287164);
}
```

Mitigation

Claiming should be made to the receiver of the funds, `to`. The `_claim()` function should be adapted to receiver an `account` argument.

Discussion

santiellena

Escalate.

This issue should be Low/Informational.

In order to "exploit" this vulnerability, the attacker has to donate a sufficient amount to the victim to move the `avgStart` parameter enough. The inability of the victim to withdraw will be temporal and not permanent because as time passes, the penalty decreases, so the issue is incorrect when states that `will lead to stuck funds`.

Additionally, as this issue is about causing temporal unavailability of part of the funds of a user, the DoS rules might apply.

```
i. The issue causes locking of funds for users for more than a week.
ii. The issue impacts the availability of time-sensitive functions (cutoff
 ↪  functions are not considered time-sensitive). If at least one of these are
 ↪  describing the case, the issue can be a Medium. If both apply, the issue can
 ↪  be considered of High severity. Additional constraints related to the issue
 ↪  may decrease its severity accordingly.
Griefing for gas (frontrunning a transaction to fail, even if can be done
 ↪  perpetually) is considered a DoS of a single block, hence only if the
 ↪  function is clearly time-sensitive, it can be a Medium severity issue.
```

Point 1) For this to cause unavailability of part of the funds due to temporal penalties for more than a week, big donations must be made which makes the attack unfeasible. Point 2) Claiming rewards is not time sensitive.

For all of the above, I believe that this issue doesn't warrant medium severity.

sherlock-admin3

> Escalate.
>
> This issue should be Low/Informational.
>
> In order to "exploit" this vulnerability, the attacker has to donate a sufficient amount to the victim to move the `avgStart` parameter enough. The inability of the victim to withdraw will be temporal and not permanent because as time passes, the penalty decreases, so the issue is incorrect when states that `will lead to stuck funds`.
>
> Additionally, as this issue is about causing temporal unavailability of part of the funds of a user, the DoS rules might apply.
>
> ```
> i. The issue causes locking of funds for users for more than a week.
> ```

```
ii. The issue impacts the availability of time-sensitive functions (cutoff
↪  functions are not considered time-sensitive). If at least one of these
↪  are describing the case, the issue can be a Medium. If both apply, the
↪  issue can be considered of High severity. Additional constraints
↪  related to the issue may decrease its severity accordingly.
Griefing for gas (frontrunning a transaction to fail, even if can be done
↪  perpetually) is considered a DoS of a single block, hence only if the
↪  function is clearly time-sensitive, it can be a Medium severity issue.
```

> Point 1) For this to cause unavailability of part of the funds due to temporal penalties for more than a week, big donations must be made which makes the attack unfeasible. Point 2) Claiming rewards is not time sensitive.
>
> For all of the above, I believe that this issue doesn't warrant medium severity.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

0xsimao

The issue is valid because:

1. It doesn't need to be an attacker, the same user can use one wallet to deposit into another, triggering this bug
2. It always leads to stuck funds for longer and directly to loss of funds in some cases

   Additionally, in case they decide to withdraw after depositing, before the rewards vest again, they will lose these funds to savings.
3. Due to 1. Making the funds stuck for a lot of time is possible and has no significant requirements (up to 24 weeks as stated in the issue)

santiellena

1) Agree. However, I still consider it borderline Medium/Low
2) Temporary unavailability is not loss of funds. In which cases loss of funds occurs?
3) As stated in the issue, there actually are significant requirements. The user has to duplicate its staking position in that deposit from other wallet.

I will keep the Escalation so the HoJ reviews it. I think both are fair points.

0xsimao

Loss of funds occurs when the user withdraws after depositing. Instead of claiming the rewards on the deposit, `avgStart` would increase without claiming. Then, when withdrawing, here, on claimWithdraw(), it transfers these rewards to savings, when they should have gone to the user. So the rewards would be lost for the user on unstake. It's not like he can unstake and claim the rewards later, a part of them or all would be lost.

santiellena

You are right @0xsimao, in case of staking and then unstaking that will lead to loss of accrued rewards. Unluckily, I didn't have time to check your point before to delete the escalation.

WangSecurity

I agree with @0xsimao arguments and believe medium severity is appropriate here. If you need a deeper analysis from my side, let me know, but I don't see a point in repeating the same points. Planning to reject the escalation and leave the issue as it is.

WangSecurity

Result: Medium Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- santiellena: rejected

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/exactly/protocol/pull/749

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-2: Attackers will reset `avgStart` of any user making rewards stuck for longer and get lost to savings

Source: https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts-judging/issues/22

Found by

0x73696d616f, HackTrace

Summary

`StakedExa::_update()` resets `avgStart[to]` whenever the time passed is bigger than the reference time. An attacker can deposit 1 wei to some user and reset the `avgStart`, which will make claiming for the `to` address return 0 rewards until at least `minTime` and/or the `to` address loses the rewards to savings when withdrawing.

Root Cause

In `StakedExa.sol::151`, `if (time > memRefTime) avgStart[to] = block.timestamp * 1e18;` resets the average start time to the current block timestamp, which may be performed by an attacker donating just 1 wei to the `to` address.

Internal pre-conditions

1. User must stake for at least `refTime`, which is expected as this the moment the user can claim the maximum rewards.

External pre-conditions

None.

Attack Path

1. User stakes until reaching at least the reference time + 1.
2. Attacker deposits 1 wei to the user forcing `avgStart` to reset and discount all rewards to 0 until `minTime` passes.

Impact

User will not receive any rewards until `minTime` and will lose to savings when withdrawing as no rewards would have vested due to `avgStart[to]` having been reset.

PoC

Add the following test to `StakedEXA.t.sol`.

```
function test_POC_anyone_can_reset_stakedTime() external {
  bool attack = true;
  uint256 assets = 1e18;
  uint256 time = refTime;

  exa.mint(address(this), assets);
  stEXA.deposit(assets, address(this));

  uint256 initTime = block.timestamp * 1e18;
  skip(time + 1);
  uint256 finalTime = block.timestamp * 1e18;

  stEXA.claim(rA);
  assertEq(stEXA.avgStart(address(this)), initTime);

  // Add some more rewards
  rA.mint(address(stEXA), initialAmount);
  stEXA.notifyRewardAmount(rA, initialAmount);

  // Attacker deposits 1 to reset start time and get rewards stuck
  if (attack) {
    address attacker = makeAddr("attacker");
    exa.mint(attacker, 1);
    vm.startPrank(attacker);
    exa.approve(address(stEXA), 1);
    stEXA.deposit(1, address(this));
    vm.stopPrank();
    assets += 1;
  }

  skip(minTime);

  assertEq(stEXA.avgStart(address(this)), attack ? finalTime : initTime,
↪   "avgStart");
  assertEq(stEXA.claimable(rA, address(this), assets), attack ? 0 :
↪   20833334711198888308, "claimable");
}
```

Mitigation

Remove the `avgStart[to]` reset mechanism in `_update()`, as users can decrease their staked time by withdrawing and depositing again, it should not be able to

reset like this because they might want to just keep claiming or withdraw in the near future while paying only some `excessFactor` rewards. With the code as is, attackers can manipulate other users rewards and make them lose funds.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/exactly/protocol/pull/750

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-3: Frozen/paused Market that is harvested from in StakedEXA will DoS deposits leading to loss of yield

Source: https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts-judging/issues/35

Found by

0x73696d616f

Summary

`StakedEXA::harvest()` withdraws from the `market` of the `StakedEXA` contract if the provider has assets there and has approved `StakedEXA`. However, the market may be paused/frozen, but `StakedEXA::harvest()` will try to withdraw anyway, which will make it revert. As it is part of the deposit flow, it means all deposits will revert and new users will not be able to collect yield.

Root Cause

In `StakedEXA::_update()`, `StakedEXA::harvest()` is called if it is a mint (`from == address(0)`), here. Then. in `StakedEXA:harvest()`, it withdraws from the `market` regardless of its state, reverting if frozen or paused.

Internal pre-conditions

None.

External pre-conditions

1. Market is frozen or paused.

Attack Path

1. Market is frozen or paused.
2. StakedEXA deposits are frozen.

Impact

Frozen `StakedEXA` deposits which means new users will miss out on the yield and current users will enjoy a guaranteed higher yield.

PoC

Add the following test to `StakedEXA.t.sol` confirming that deposits are halted when the market is paused.

```
function test_POC_halted_deposits() external {
  uint256 assets = 1e18;

  market = Market(address(new ERC1967Proxy(address(new
↪  Market(ERC20Solmate(address(providerAsset)), new Auditor(18))), "")));
  market.initialize(
    "STEXA",
    3,
    1e18,
    new InterestRateModel(
      IRMParameters({
        minRate: 3.5e16,
        naturalRate: 8e16,
        maxUtilization: 1.1e18,
        naturalUtilization: 0.75e18,
        growthSpeed: 1.1e18,
        sigmoidSpeed: 2.5e18,
        spreadFactor: 0.2e18,
        maturitySpeed: 0.5e18,
        timePreference: 0.01e18,
        fixedAllocation: 0.6e18,
        maxRate: 15_000e16
    }),
    Market(address(0))),
    0.02e18 / uint256(1 days),
    1e17,
    0,
    0.0046e18,
    0.42e18
  );
  market.grantRole(market.PAUSER_ROLE(), address(this));

  vm.prank(address(stEXA));
  providerAsset.approve(address(market), type(uint256).max);

  stEXA.setMarket(market);

  providerAsset.mint(PROVIDER, 1_000e18);
  vm.startPrank(PROVIDER);
  providerAsset.approve(address(market), type(uint256).max);
  market.deposit(1_000e18, PROVIDER);
  market.approve(address(stEXA), 1_000e18);
```

```
  vm.stopPrank();

  market.pause();

  exa.mint(address(this), assets);

  vm.expectRevert();
  stEXA.deposit(assets, address(this));
}
```

Mitigation

Check if the market is paused or frozen and do not harvest if this is the case.

Discussion

santipu03

Escalate

I believe this issue should be low severity according to the following rule from the Sherlock docs:

> An admin action can break certain assumptions about the functioning of the code. Example: Pausing a collateral causes some users to be unfairly liquidated or any other action causing loss of funds. This is not considered a valid issue.

sherlock-admin3

> Escalate
>
> I believe this issue should be low severity according to the following rule from the Sherlock docs:
>
> > An admin action can break certain assumptions about the functioning of the code. Example: Pausing a collateral causes some users to be unfairly liquidated or any other action causing loss of funds. This is not considered a valid issue.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

0xsimao

The rule doesn't apply here because the concerns are in different contracts. It's not `StakedEXA` that is paused, but the `Market` integration itself. `StakedEXA` should keep working normally.

WangSecurity

As I understand `market` is external and it's the external admin pausing the market, not Exactly admin, correct? If the market is frozen/paused, why the user should be able to deposit into that market?

0xsimao

@WangSecurity the market is also a contract of the Exactly protocol. The relevant part is that each contract, `Market` and `StakedEXA`, have their own admin and pausing functionalities. Harvesting is one functionality of `StakedEXA` and it should not DoS deposits, as `StakedEXA` fully works regardless of the market's state. `StakedEXA` has its own pausing functionality here.

Answering your question, the purpose of `StakedEXA` is staking the `EXA`, the selected market is just chosen to get the rewards from the provider to the users staking, the `EXA` staked is not actually deposited into the market. Thus it's crucial users can continue staking even if the current selected market for harvesting gets paused/frozen. Rewards are still being accumulated from previous harvests (the duration is 24 weeks). Thus, it would make no sense to forcibly DoS staking as users should be able to deposit and benefit from this yield and the admin of `StakedEXA` should set a new market or wait for the market to be live again, not stop staking.

WangSecurity

Can you share where you got this information from? I mean the fact that the users must be able to deposit at any time even if the market is paused/frozen, is it from README, code comments or docs? I don't see it in the first two, but I might be missing something?

0xsimao

@WangSecurity EXA that is deposited in StakedEXA is not deposited in the market, it stays in the contract and accumulates rewards from the provider. The provider is the one that deposits in the market but this is external to StakedEXA. The contract is meant to lock liquidity and benefit users that hold their EXA.

It would make sense to freeze deposits if a market is frozen if the EXA tokens were deposited to the market when deposited in StakedEXA, but it is not the case.

Feel free to go through the StakedEXA code, nowhere does it deposit user deposits in the market. It only deposits the provider funds in the market to the savings when the provider gives allowance, but user EXA deposits stay in StakedEXA. Thus, as

the EXA funds are not deposited in the market, deposits should not be DoSed when the market is frozen.

WangSecurity

@0xsimao as I understand, you mean that users should be able to deposit into StakedEXA at any moment in time, correct? Can you share what this understanding is based on? Is it documented or based on how the code functions (excuse me if a silly question, just don't see it in the code comments or README)?

0xsimao

@WangSecurity yes, that is correct.

The harvesting functionality should not DoS staking ever, there may be periods without new rewards from `harvest()`, the `provider` chooses when/how much rewards to distribute. Take a look at this comment here, which says: `This function withdraws the maximum allowable assets from the provider's market`, If the market is paused, the maximum allowable assets should be 0. However, it tries to withdraw anyway and reverts.

WangSecurity

Thank you for these clarifications. I see how the concern about admin actions has got here. But, indeed, the users should be able to deposit in StakedEXA under any circumstances and when depositing into StakedEXA, users' funds indeed stay inside the contract and are not deposited inside the market. Additionally, both Market and StakedEXA have different admins and pausing functionalities. That's why I agree the rule shouldn't apply here and the issue should remain a valid medium. Planning to reject the escalation and leave the issue as it is.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- santipu03: rejected

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/exactly/protocol/pull/751

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-4: Setting a new market will make depositing to the market impossible when harvesting, DoSing deposits

Source: https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts-judging/issues/41

Found by

0x73696d616f, 0xBugHunter, blockchain555, eeyore

Summary

In `StakedEXA::harvest()`, it deposits to the `market` the resulting savings that are not notified as rewards from the provider. The approval of the asset of the `Market` to the `Market` is only performed in the constructor, which means that if a new `Market` is set via `StakedEXA::setMarket()`, it currently does not approve it and will DoS deposits and make it impossible to correctly switch markets.

Root Cause

In `StakedEXA.sol::460`, `setMarket()`, a new market is set but an approval is not given of the `market.asset()` to the `market`, which will DoS deposits.

Internal pre-conditions

1. A new market is set.

External pre-conditions

None.

Attack Path

1. `StakedEXA::setMarket()` is called, setting a new market.
2. Deposits are DoSed because it tries to deposit to the new market without approving it first and reverts.

Impact

DoSed deposits and impossability of correctly changing markets.

PoC

```
function test_POC_DoSedDeposits_DueToSettingANewMarket() external {
  bool triggerRevert = true;

  uint256 assets = 1e18;
  market = Market(address(new ERC1967Proxy(address(new
↪  Market(ERC20Solmate(address(providerAsset)), new Auditor(18))), "")));
  market.initialize(
    "STEXA",
    3,
    1e18,
    new InterestRateModel(
      IRMParameters({
        minRate: 3.5e16,
        naturalRate: 8e16,
        maxUtilization: 1.1e18,
        naturalUtilization: 0.75e18,
        growthSpeed: 1.1e18,
        sigmoidSpeed: 2.5e18,
        spreadFactor: 0.2e18,
        maturitySpeed: 0.5e18,
        timePreference: 0.01e18,
        fixedAllocation: 0.6e18,
        maxRate: 15_000e16
    }),
    Market(address(0))),
    0.02e18 / uint256(1 days),
    1e17,
    0,
    0.0046e18,
    0.42e18
  );
  market.grantRole(market.PAUSER_ROLE(), address(this));

  // FIX market change
  if (!triggerRevert) {
    vm.prank(address(stEXA));
    providerAsset.approve(address(market), type(uint256).max);
  }

  stEXA.setMarket(market);

  providerAsset.mint(PROVIDER, 1_000e18);

  vm.startPrank(PROVIDER);
```

```
    providerAsset.approve(address(market), type(uint256).max);
    market.deposit(1_000e18, PROVIDER);
    market.approve(address(stEXA), 1_000e18);
    vm.stopPrank();

    exa.mint(address(this), assets);

    if (triggerRevert) vm.expectRevert();
    stEXA.deposit(assets, address(this));
  }
```

Mitigation

Do the same as in the constructor and approve the new `Market` when `setMarket()` is called with `type(uint256).max` `market.asset()` assets.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/exactly/protocol/pull/748

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-5: Market utilization ratio near 100% will DoS deposits as harvest tries to withdraw and reverts

Source: https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts-judging/issues/45

Found by

0x73696d616f, santipu_

Summary

`StakedEXA::_update()` harvests when depositing, withdrawing assets from the provider in the given market. However, due to the utilization ratio check in the market, it may not be possible to withdraw these assets, reverting. This will halt deposits and secure some extra yield for users that potentially make this happen.

Root Cause

In `StakedEXA::356` it withdraws without checking if there is enough protocol liquidity in the market such that it is possible to withdraw. If there is too much debt, it will revert and halt deposits.

Internal pre-conditions

None.

External pre-conditions

1. Market needs to be close to maximum utilization ratio.

Attack Path

1. Users borrow a lot from the underlying market, making the utilization ratio reach 100%
2. Users try to deposit but revert due to trying to withdraw assets from the provider in the market that would leave the market with less deposits than borrows.

Impact

Deposits are DoSed leading to extra yield for current depositors and loss of yield for future ones that can't stake.

PoC

Add the following test to `StakedEXA.t.sol` as proof.

```
function test_POC_DoSedDeposits_FailsAsMarketIsFullyUtilized() external {
  Auditor auditor = Auditor(address(new ERC1967Proxy(address (new Auditor(18)),
↪  "")));

  uint256 assets = 1e18;
  market = Market(address(new ERC1967Proxy(address(new
↪  Market(ERC20Solmate(address(providerAsset)), auditor)), "")));
  market.initialize(
    "STEXA",
    3,
    1e18,
    new InterestRateModel(
      IRMParameters({
        minRate: 3.5e16,
        naturalRate: 8e16,
        maxUtilization: 1.1e18,
        naturalUtilization: 0.75e18,
        growthSpeed: 1.1e18,
        sigmoidSpeed: 2.5e18,
        spreadFactor: 0.2e18,
        maturitySpeed: 0.5e18,
        timePreference: 0.01e18,
        fixedAllocation: 0.6e18,
        maxRate: 15_000e16
    }),
    Market(address(0))),
    0.02e18 / uint256(1 days),
    1e17,
    0,
    0.0046e18,
    0.42e18
  );

  auditor.initialize(Auditor.LiquidationIncentive(0, 0));
  auditor.enableMarket(market, MockPriceFeed(auditor.BASE_FEED()), 0.9e18);

  // FIX market change, another issue
  vm.prank(address(stEXA));
  providerAsset.approve(address(market), type(uint256).max);

  stEXA.setMarket(market);

  providerAsset.mint(PROVIDER, 1_000e18);
```

```
  vm.startPrank(PROVIDER);
  providerAsset.approve(address(market), type(uint256).max);
  market.deposit(1_000e18, PROVIDER);
  market.approve(address(stEXA), 1_000e18);
  vm.stopPrank();

  address user = makeAddr("user");
  providerAsset.mint(user, 1_000e18);
  vm.startPrank(user);
  providerAsset.approve(address(market), type(uint256).max);
  market.deposit(1_000e18, user);
  market.borrow(1_000e18*90*90/100/100, user, user);
  vm.stopPrank();

  // Random amount of time to make sure borrowed amount reaches deposits
  skip(1000 weeks);

  exa.mint(address(this), assets);

  vm.expectRevert();
  stEXA.deposit(assets, address(this));
}
```

Mitigation

Cap the amount to withdraw to the maximum amount that does not revert, that is, the amount that makes the utilization ratio reach 100% but not over it.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/exactly/protocol/pull/751

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-6: Anyone will DoS setting a new rewards duration which harms the protocol/users as they will receive too much or too little rewards

Source: https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts-judging/issues/46

Found by

0x73696d616f

Summary

Rewards are destributed via `StakedEXA::notifyRewardAmount()` over a certain duration and amount, which originates a rate. The protocol may intend to change this rate by altering the duration, making users receive more or less rewards depending on intent. The duration of the rewards is set in `StakedEXA::setRewardsDuration()`, but it will be near impossible to set as there is an automatic mechanism to harvest rewards from the provider.

Root Cause

In `StakedEXA::443`, it's not possible to set a new duration if the current rewards are still not finished. This will always be the case as the automated provider harvesting will keep notifying new rewards and resetting the finish time of the rewards.

Internal pre-conditions

None.

External pre-conditions

None.

Attack Path

1. Provider harvests new rewards.
2. `StakedEXA::harvest()` is called, which notifies new rewards and updates the finish time.
3. `StakedEXA::setRewardsDuration()` reverts as `block.timestamp` has not reached `rewardData.finishAt`.

Impact

The ability to change the duration of rewards is DoSed which will impact the amount of rewards users get over time.

PoC

The function `StakedEXA::setRewardsDistribution()` shows that it's not possible to update the duration if the current reward period is not finished:

```
function setRewardsDuration(IERC20 reward, uint40 duration) public
 ↪  onlyRole(DEFAULT_ADMIN_ROLE) {
  RewardData storage rewardData = rewards[reward];
  if (rewardData.finishAt > block.timestamp) revert NotFinished(); //@audit DoS
 ↪  this

  rewardData.duration = duration;

  emit RewardsDurationSet(reward, msg.sender, duration);
}
```

`StakedEXA::harvest()` notifies new rewards, which updates the current reward period finish, making it impossible to change the duration.

Mitigation

The duration can be set by carefully adjusting the current reward rate to reflect the new duration. One example solution is doing:

```
function setRewardsDuration(uint256 _rewardsDuration) external
 ↪  onlyRole(DEFAULT_ADMIN_ROLE) {
    uint256 periodFinish_ = periodFinish;
    if (block.timestamp < periodFinish_) {
        uint256 leftover = (periodFinish_ - block.timestamp) * rewardRate;
        rewardRate = leftover / _rewardsDuration;
        periodFinish = block.timestamp + _rewardsDuration;
    }

    rewardsDuration = _rewardsDuration;
    emit RewardsDurationUpdated(rewardsDuration);
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/exactly/protocol/pull/753

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-7: Having no deposits in `StakedEXA` will lead to stuck rewards when harvesting

Source: https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts-judging/issues/48

Found by

0x73696d616f, KupiaSec

Summary

The provider in `StakedEXA` is free to distributed rewards as it pleases and there is an automated way to distribute via `StakedEXA::harvest()`. The harvest setup may be started but no actual deposits have been made to `StakedEXA` or users may withdraw from `StakedEXA` while automated ongoing rewards are happening, which will lead to stuck rewards inside `StakedEXA`.

Root Cause

In `StakedEXA::globalIndex()` it returns a stale index if the `totalSupply()` is null (no deposits), but it should instead calculate the amount (`rate x deltaTime`) and send it to savings. This way rewards will never be stuck.

Internal pre-conditions

1. No deposits are present in `StakedEXA`.

External pre-conditions

None.

Attack Path

1. Deposits were not yet made and rewards are harvested or users withdraw all deposits while rewards are rolling out.

Impact

Stuck rewards as the index is not increased but `rewardData.updatedAt` increases.

PoC

In `StakedEXA::updateIndex()`, the index is updated to `globalIndex(reward);` and `rewardData.updatedAt` to the current `block.timestamp` or `rewardData.finishAt`.

```
function updateIndex(IERC20 reward) internal {
  RewardData storage rewardData = rewards[reward];
  rewardData.index = globalIndex(reward);
  rewardData.updatedAt = uint40(lastTimeRewardApplicable(rewardData.finishAt));
}
```

In `StakedEXA::globalIndex()`, the index is not increased if the total supply is null:

```
function globalIndex(IERC20 reward) public view returns (uint256) {
  RewardData storage rewardData = rewards[reward];
  if (totalSupply() == 0) return rewardData.index;

  return
    rewardData.index +
    (rewardData.rate * (lastTimeRewardApplicable(rewardData.finishAt) -
↪   rewardData.updatedAt)).divWadDown(
      totalSupply()
    );
}
```

Thus, as can be seen, the index will not be updated by `rewardData.updatedAt` is increased, losing these rewards forever.

Mitigation

If the total supply is null the amount not distributed can be calculated by doing `rate x deltaTime` and send to savings.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/exactly/protocol/pull/752

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-8: Liquidating maturies with unassigned earnings will not take into account floating assets increase leading to loss of funds

Source: https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts-judging/issues/64

The protocol has acknowledged this issue.

Found by

0x73696d616f

Summary

`Market::liquidate()` calculates the amount to liquidate in `Auditor::checkLiquidation()`, which does not take into account floating assets accrual due to unassigned earnings in `Market::noTransferRepayAtMaturity()`. Thus, it will underestimate the collateral a certain account has and liquidate less than it should, either giving these funds to the liquidatee or in case the debt is bigger than the collateral, leave this extra untracked collateral in the account which will not enable `clearBadDebt()` to work.

Root Cause

In `Auditor::checkLiquidation()`, the collateral is computed just by looking at the shares of an account, without taking into account maturities accruing unassigned earnings. In `Market::noTransferRepayAtMaturity()`, maturities are repaid and floating assets are increased due to backup earnings. Thus, when it seizes the collateral, it will seize at most the collateral without taking into account the new amount due to the floating assets sudden increase, as `Auditor::checkLiquidation()` is called prior the floating assets increase.

Internal pre-conditions

1. Pool has unassigned earnings and user is liquidated.

External pre-conditions

None.

Attack Path

1. Users borrow at maturity and generate unassigned earnings.

2. One of these users is liquidated but the collateral preview is incorrect and the liquidation is incomplete and the funds are lost.

Impact

If the user has less debt than collateral but the debt would require most of its collateral, it would not be possible to seize it all. If the user has more debt than collateral a portion of the collateral will be leftover which will not allow bad debt to be cleared right away via `clearBadDebt()` and stays accruing.

PoC

Add the following test to `Market.t.sol`, confirming that the user should have all its collateral seized given that debt > collateral but some collateral remains due to the increase in floating assets.

```
function test_POC_FloatingAssetsIncrease() external {
  uint256 maxVal = type(uint256).max;

  market.deposit(10000 ether, address(this));

  vm.startPrank(BOB);
  market.deposit(100 ether, BOB);
  market.borrowAtMaturity(4 weeks, 60e18, maxVal, BOB, BOB);
  vm.stopPrank();

  skip(10 weeks);

  (uint256 coll, uint256 debt) = market.accountSnapshot(BOB);
  assertEq(coll, 100e18);
  assertEq(debt, 111.246904109483404820e18);

  vm.startPrank(ALICE);
  market.liquidate(BOB, 100_000 ether, market);
  vm.stopPrank();

  (coll, debt) = market.accountSnapshot(BOB);
  assertEq(coll, 4.557168045571680e15);
  assertEq(debt, 20.337813200392495730e18);
}
```

Mitigation

`Auditor::checkLiquidation()` should compute the increase in collateral by previewing the floating assets increase due to accrued unassigned earnings in a

fixed pool or instead of increasing floating assets these earnings could increase the earnings accumulator.

Issue M-9: Rewards Can Be Harvested Even When Distribution Is Marked As Finished

Source: https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts-judging/issues/66

Found by

KupiaSec, sakshamguruji

Summary

In the contract StakedEXA.sol the finishDistribution() intends to finish the distribution of the reward token , but it fails to do so. Rewards can still be harvested from the Market and later claimed , meaning , rewards can be distributed violating the finish time of the distribution.

Vulnerability Detail

1.) The `finishDistribution()` intends to finish the distribution of the reward token , let's assume the admin now wants to finish the distribution , and calls the function -->

https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts/blob/main/protocol/contracts/StakedEXA.sol#L425

Now the finish time is set to the `block.timestamp`

2.) All the above function does is transfer reward token to savings address (dependent upon rate and time difference)

3.) After the `finishDistribution()` , a normal user can still trigger the `harvest()` function -->

https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts/blob/main/protocol/contracts/StakedEXA.sol#L344

This would firstly calculate `assets` to be withdrawn from the market which is dependent on the allowance of the distributor and then the assets are withdrawn from the market. And then `notifyRewardAmount()` is triggered at the last.

https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts/blob/main/protocol/contracts/StakedEXA.sol#L209

4.) `notifyRewardAmount()` the reward.rate would be calculated as

```
RewardData storage rewardData = rewards[reward];
    if (block.timestamp >= rewardData.finishAt) {
      rewardData.rate = amount / rewardData.duration;
    }
```

Because the finishAt has been marked to the timestamp when finaliseRewards was called.

5.) And lastly `rewardsData` is updated

```
rewardData.finishAt = uint40(block.timestamp) + rewardData.duration;
    rewardData.updatedAt = uint40(block.timestamp);
```

6.) Now the user can claim the rewards since they are harvested , triggering the `claim()` function

7.) Even after finalisation of rewards , the reward tokens were distributed from the market and harvested and notified. It can be seen as if provider lost those assets which were not meant to be distributed.

Impact

Rewards still being able to be harvested even after the finalisation of rewards , the provider would loose his assets if had given a infinite approval . The finalise rewards marks the `Finishes the distribution of a reward token`

Code Snippet

https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts/blob/main/protocol/contracts/StakedEXA.sol#L425-L432

Tool used

Manual Review

Recommendation

The way to avoid starting a new distribution would be for the provider to set 0 allowances on the market or withdraw the assets

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/exactly/protocol/pull/753

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-10: Liquidations will leave dust when repaying expired maturities, making it impossible to clear bad debt putting the protocol at a risk of insolvency

Source: https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts-judging/issues/69

The protocol has acknowledged this issue.

Found by

0x73696d616f

Summary

`Market::liquidate()` gets the maximum assets to liquidate from `Auditor::checkLiquidation()`. Then, when repaying maturities, when they have expired, it calculates how much principal of the maturity it has to liquidate to take into account the penalty rate and liquidate at most `maxRepayAssets`. However, there is rounding in this process that means the actual repay assets will be smaller than `maxRepayAssets`, which means the liquidatee will have dust collateral. Thus, it will be impossible to clear the bad debt, even if liquidated again later.

Root Cause

In `Market::liquidate()`, it calculates the actual repay has:

```
uint256 debt = position + position.mulWadDown((block.timestamp - maturity) *
↪  penaltyRate);
actualRepay = debt > maxAssets ? maxAssets.mulDivDown(position, debt) :
↪  maxAssets;
```

When `debt > maxAssets`, it will round down in the calculation, liquidating less assets than the maximum and leaving the liquidatee with some dust making it impossible to clear the bad debt.

Internal pre-conditions

1. Liquidatee has more debt than collateral and expired maturities.

External pre-conditions

None.

Attack Path

1. Liquidatee borrows at maturity and lets it expire, not repaying the debt until it grows bigger than the collateral.
2. Liquidator liquidates but it is never possible to fully clear the collateral so `clearBadDebt()` will never be called.

Impact

When there are expired maturities and the debt is bigger than the collateral `clearBadDebt()` will not be called, not even if liquidated successively.

PoC

Place the following test in `Market.t.sol`. Notice how it reverts due to `ZERO_WITHDRAW` as the `actualRepayAssets` is zero.

```
function test_POC_ClearBadDebt_Impossible_DueToDust() external {
  uint256 maxVal = type(uint256).max;

  market.deposit(10000 ether, address(this));

  vm.startPrank(BOB);
  market.deposit(100 ether, BOB);
  market.borrowAtMaturity(4 weeks, 60e18, maxVal, BOB, BOB);
  vm.stopPrank();

  vm.startPrank(ALICE);
  market.deposit(10, ALICE);
  market.borrowAtMaturity(4 weeks, 1, maxVal, ALICE, ALICE);
  vm.stopPrank();

  skip(20 weeks);

  (uint256 coll, uint256 debt) = market.accountSnapshot(BOB);
  console.log(coll);
  console.log(debt);

  vm.startPrank(ALICE);
  market.liquidate(BOB, 100_000 ether, market);
  vm.stopPrank();
  (coll, debt) = market.accountSnapshot(BOB);
  console.log(coll);
  console.log(debt);

  vm.startPrank(ALICE);
```

```
    market.liquidate(BOB, 100_000 ether, market);
    vm.stopPrank();
    (coll, debt) = market.accountSnapshot(BOB);
    console.log(coll);
    console.log(debt);

    vm.startPrank(ALICE);
    vm.expectRevert(); // ZERO WITHDRAW
    market.liquidate(BOB, 100_000 ether, market);
    vm.stopPrank();
}
```

Mitigation

The mentioned rounding error is hard to deal with, it's easier to set some threshold to clear bad debt so rounding errors can be disregarded.

Issue M-11: Liquidator will leave a pool with unassigned earnings on `Market::clearBadDebt()` free to claim for anyone when the repaid maturity is not the last

Source: https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts-judging/issues/73

Found by

0x73696d616f

Summary

This issue remains from the last audit, only a partial fix was applied.

The issue is that in `Market::clearBadDebt()`, when a maturity is repaid, the unassigned earnings of this pool only go to the earnings accumulator when the last maturity is liquidated. However, if it's not the last maturity, the problem remains in case there is another borrowed maturity, and some user can claim these unassigned earnings just by supplying a few funds. Or if all `floatingBackupBorrowed` was due to the repaid maturity, and now it is 0, so users can borrow and deposit 1 wei to steal the unassigned earnings.

Root Cause

In `Market:661`, the `earningsAccumulator is only increased [when](https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts/blob/main/protocol/contracts/Market.sol#L652-L655)` fixedPools[maturity].borrowed == position.principal. position.principal`is always 0, as`fixedBorrowPositions[maturity][borrower]`has been deleted. Thus it will only assign the unassigned earnings when`fixedPools[maturity].borrowed‘ becomes null after a repayment. This means that as long as there are other borrowed maturities, the unassigned earnings may be stolen.

Internal pre-conditions

1. Unassigned earnings must exist.
2. Debt of fixed borrows has to exceed user's collateral.

External pre-conditions

None.

Attack Path

1. Deposit funds to `Market` via `Market::deposit()` for the maturities to borrow, increasing `floatingBackupBorrow` and generating unassigned earnings.
2. 1 maturity is borrowed which will be liquidated.
3. At least 1 other maturity is borrowed so after liquidating the first, `fixedPools[maturity].borrowed` is not null.
4. Liquidate one of the maturities, so the unassigned earnings resulting from this maturity will not go to the `earningsAccumulator`.
5. Deposit at maturity to capture the unassigned earnings from both maturities, while only contributing liquidity to the other non liquidated one, which may be a dust amount, but the full unassigned earnings are captured to a single user.

Impact

Attacker can claim a lot of unassigned earnings just by depositing at maturity enough (may be as small as 1 wei) to cover remaining `floatingBackupBorrowed`.

PoC

Add the following POC to `Market.t.sol`, showing how a depositMaturity of 10 wei is enough to steal all unassigned maturities.

```
function test_POC_ClearBadDebt_StealMaturities() external {
  uint256 maxVal = type(uint256).max;

  marketWETH.deposit(100000 ether, address(this));
  marketWETH.borrowAtMaturity(4 weeks, 10000e18, maxVal, address(this),
↪   address(this));
  marketWETH.repayAtMaturity(4 weeks, maxVal, maxVal, address(this));

  vm.startPrank(BOB);
  market.deposit(100 ether, BOB);
  auditor.enterMarket(market);
  marketWETH.borrowAtMaturity(4 weeks, 60e18, maxVal, BOB, BOB);
  vm.stopPrank();

  vm.startPrank(ALICE);
  weth.mint(ALICE, 1_000_000 ether);
  weth.approve(address(marketWETH), type(uint256).max);
  marketWETH.deposit(10, ALICE);
  marketWETH.borrowAtMaturity(4 weeks, 1, maxVal, ALICE, ALICE);
  vm.stopPrank();
```

```
  daiPriceFeed.setPrice(0.6e18);

  (uint256 coll, uint256 debt) = marketWETH.accountSnapshot(BOB);
  console.log(debt);
  (coll, debt) = market.accountSnapshot(BOB);
  console.log(coll);

  vm.startPrank(ALICE);
  marketWETH.liquidate(BOB, 100_000 ether, market);
  vm.stopPrank();

  (, , uint256 unassignedEarningsAfter, ) = marketWETH.fixedPools(4 weeks);
  assertEq(unassignedEarningsAfter, 450285328874794 52);

  address attacker = makeAddr("attacker");
  vm.startPrank(attacker);
  deal(address(weth), attacker, 10);
  weth.approve(address(marketWETH), 10);
  marketWETH.depositAtMaturity(4 weeks, 10, 0, attacker);
  vm.stopPrank();

  (, , unassignedEarningsAfter, ) = marketWETH.fixedPools(4 weeks);
  assertEq(unassignedEarningsAfter, 0);

  // Attacker gets all the unassigned earnings by contributing only 10
  (uint256 principal, uint256 fee) = marketWETH.fixedDepositPositions(4 weeks,
↪   attacker);
  assertEq(principal, 10);
  assertEq(fee, 40525679598731507);
}
```

Mitigation

The unassigned earnings should be assigned to the earnings accumulator pro-rata to the `floatingBackupBorrowed` cleared in the bad debt clearance.

Discussion

santichez

Hi @0xsimao , the mitigation sentence seems a bit unclear. Could you please provide more detail or clarify it further? Thank you.

0xsimao

Hey @santichez, smth like this, but the yield goes to the earnings accumulator.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/exactly/protocol/pull/755

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-12: Some bad debt will not be cleared when it should which will cause accrual of bad debt decreasing the protocol's solvency

Source: https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts-judging/issues/74

The protocol has acknowledged this issue.

Found by

0x73696d616f

Summary

In `Market::clearBadDebt()`, it only clears bad debt if the `earningsAccumulator` is bigger than the bad debt to clear, but the fixed pool in question may have unassigned earnings which would be added to the earnings accumulator. Thus, the accumulator would have funds to handle the bad debt, but due to checking if it is enough before taking into account its increase, it will not clear the bad debt and will decrease protocol solvency.

Root Cause

In `Market:641`, it first checks if `accumulator >= badDebt` and then in `Market:653` it increases the accumulator. This should be done in the opposite order as the accumulator may actual be big enough with the added unassigned earnings to clear the bad debt.

Internal pre-conditions

1. Fixed pool has unassigned earnings.
2. User is liquidated.
3. Accumulator is bigger than the bad debt if it takes into account unassigned earnings.

External pre-conditions

None.

Attack Path

1. User borrows maturities but lets them expire and accrues more debt than collateral.
2. User is liquidated, but the accumulator is not enough to cover the bad debt without the unassigned earnings increase, so bad debt keeps accruing.

Impact

Bad debt accrual which harms protocol users as it increases the risk of insolvency.

PoC

The following code snippet can be verified to confirm the issue:

```
function clearBadDebt(address borrower) external {
  {
    {
      ...
      if (accumulator >= badDebt) {
        ...
        if (fixedPools[maturity].borrowed == position.principal) {
          earningsAccumulator += fixedPools[maturity].unassignedEarnings;
          fixedPools[maturity].unassignedEarnings = 0;
        }
        ...
      }
    }
  }
  ...
}
```

Mitigation

Increase the earnings accumulator first and only then compare it against the bad debt.

Issue M-13: Precision Loss in `notifyRewardAmount` Function Causes Unclaimable RewardToken

Source: https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts-judging/issues/96

Found by

0uts1der, 0x73696d616f, Bigsam, HaxSecurity, KupiaSec, santiellena, santipu_

Summary

In the `StakedEXA` contract, the `notifyRewardAmount` function suffers from precision loss when calculating the reward rate, leading to some rewards being locked and unclaimable.

Vulnerability Detail

In the `StakedEXA` contract, there is a precision loss in the `notifyRewardAmount` function when calculating `rewardData.rate`, which results in some of the reward funds being locked in the contract and not being available for distribution. This leads to economic loss.

```
function notifyRewardAmount(IERC20 reward, uint256 amount, address notifier)
↪   internal onlyReward(reward) {
    updateIndex(reward);
    RewardData storage rewardData = rewards[reward];
    if (block.timestamp >= rewardData.finishAt) {
      rewardData.rate = amount / rewardData.duration;
    } else {
      uint256 remainingRewards = (rewardData.finishAt - block.timestamp) *
↪   rewardData.rate;
      rewardData.rate = (amount + remainingRewards) / rewardData.duration;
    }

    if (rewardData.rate == 0) revert ZeroRate();
    if (
      rewardData.rate * rewardData.duration >
      reward.balanceOf(address(this)) - (address(reward) == asset() ?
↪   totalAssets() : 0)
    ) revert InsufficientBalance();

    rewardData.finishAt = uint40(block.timestamp) + rewardData.duration;
    rewardData.updatedAt = uint40(block.timestamp);
```

```
    emit RewardAmountNotified(reward, notifier, amount);
  }
```

Clearly, the formulas `rewardData.rate = amount / rewardData.duration;` and `rewardData.rate = (amount + remainingRewards) / rewardData.duration;` in the calculation of `rewardData.rate` cause precision loss. This results in the final reward amount `rewardData.rate * rewardData.duration` being less than the `amount` actually passed to the `notifyRewardAmount` function. Since there is no suitable function to extract this remaining portion of funds, it causes economic loss.

A Proof of Concept (POC) can be constructed in `StakedEXA.t.sol` as follows.

```
function testRemaining() external {
    MockERC20 WBTC = new MockERC20("Wrapped BTC", "WBTC", 8);
    vm.label(address(WBTC), "WBTC");
    WBTC.mint(address(stEXA), 10e8);
    stEXA.enableReward(WBTC);
    stEXA.setRewardsDuration(WBTC, 1 weeks);
    stEXA.notifyRewardAmount(WBTC, 10e8);

    address Alice = address(0x1234);
    vm.startPrank(Alice);
    exa.mint(address(Alice), 1 ether);
    exa.approve(address(stEXA), 1 ether);
    stEXA.deposit(1 ether, Alice);

    skip(30 weeks);

    stEXA.claim(WBTC);

    emit log_named_decimal_uint("balance", WBTC.balanceOf(address(stEXA)),
↪   WBTC.decimals());
    emit log_named_decimal_uint("Alice reward WBTC", WBTC.balanceOf(Alice),
↪   WBTC.decimals());
    emit log_named_decimal_uint("saving WBTC", WBTC.balanceOf(SAVINGS),
↪   WBTC.decimals());
}
```

The output is as follows:

```
Logs:
  balance: 0.00265600
  Alice reward WBTC: 8.99760960
  saving WBTC: 0.99973440
```

Assuming WBTC as the reward token with an amount of 10 and `RewardsDuration` set

to one week, and Alice as the only staker with a stake of 1 ether worth of EXA tokens. After 30 weeks, based on the set parameters, the reward distribution should be complete. When Alice calls the `claim` function, the full 10 ether worth of rewards should be distributed to Alice and savings. However, due to precision loss, 0.0026 WBTC will remain in the contract (worth over 100 USD).

Impact

The precision loss in the `StakedEXA` contract's `notifyRewardAmount` function results in a portion of the reward funds being locked in the contract and unavailable for distribution.

Code Snippet

https://github.com/sherlock-audit/2024-07-exactly-stacking-contracts/blob/main/protocol/contracts/StakedEXA.sol#L209-L229

Tool used

Manual Review

Recommendation

Add an admin function to extract the reward tokens that remain undistributed due to precision loss.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/exactly/protocol/pull/754

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.