SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Covalent
Prepared by: Sherlock
Lead Security Expert: cergyk
Dates Audited: January 22 - January 26, 2024
Prepared on: March 1, 2024

Introduction

The Ethereum Wayback Machine - Discover how the Covalent Network is safeguarding and enhancing Ethereum's historical data availability.

Scope

Repository: covalenthq/cqt-staking

Branch: eth_migration2

Commit: 80a254a3a57e6cb7983aa057d2f77877e296806e

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
8	0

Issues not fixed or acknowledged

Medium	High
0	0

Issue M-1: Validator cannot set new address if more than 300 unstakes in it's array

Source: https://github.com/sherlock-audit/2023-11-covalent-judging/issues/25

The protocol has acknowledged this issue.

Found by

PUSH0, SadBase, alexbabits, bitsurfer, nobody2018, thank_you, yujin718

Summary

If a validator has more than 300 accumulated unstakes associated with it, then it cannot set a new address for itself. The only way to decrease the length of the `Unstaking` array is through the `setValidatorAddress()` function, but will revert if it's array is longer than 300 entries. A malicious delegator could stake 1,000 tokens, and then unstake 301 times with small amounts to fill up the unstaking array, and there is no way to remove those entries from the array. Every time `_unstake()` is called, it pushes another entry to it's array for that validator.

A malicious validator could also set it's new address to another validator, forcing a merge of information to the victim validator. The malicious validator can do this even when it's disabled with 0 tokens. So it could get to 300 length, and then send all those unstakes into another victim validators unstaking array. This is the same kind of attack vector, but should be noted that validators can assign their address to other validators, effectively creating a forceful merging of them.

The README.md suggests there is a mechanism to counteract this: "In case if there are more than 300 unstakings, there is an option to transfer the address without unstakings." But there appears to be no function in scope that can transfer the address of the validator without unstakings, or any other function that can reduce the unstakings array at all.

Vulnerability Detail

See Summary

Impact

Validator can be permanently stuck with same address if there are too many entries in it's `Unstaking` array.

Code Snippet

Validator & Unstaking Structs: https://github.com/sherlock-audit/2023-11-covalent/blob/main/cqt-staking/contracts/OperationalStaking.sol#L35-#L51 `setValidatorAddress()` length check: https://github.com/sherlock-audit/2023-11-covalent/blob/main/cqt-staking/contracts/OperationalStaking.sol#L702 deletion of array inside `setValidatorAddress()`: https://github.com/sherlock-audit/2023-11-covalent/blob/main/cqt-staking/contracts/OperationalStaking.sol#L707

Tool used

Manual Review

Recommendation

Consider having a way to set a new address without unstakings, or allow for smaller batches to be transferred during an address change if there are too many. Consider disallowing validators from changing their address to other validators.

Discussion

sudeepdino008

This is interesting. I do have few questions though

A malicious validator could also set it's new address to another validator, forcing a merge of information to the victim validator. The malicious validator can do this even when it's disabled with 0 tokens. So it could get to 300 length, and then send all those unstakes into another victim validators unstaking array. This is the same kind of attack vector, but should be noted that validators can assign their address to other validators, effectively creating a forceful merging of them.

How is setValidatorAddress operates within a `Validator` storage struct, which is retrieved from a query `_validators[validatorId]`. So each validatorId maintains its own unstakings etc. Even if a malicious validator sets its validator address to another validator, the new unstakings are maintained within the current validatorId's Validator instance (which is different from the victim validatorId and therefore victim's Validator instance and its unstakings).

Furthermore setting the newAddress to an existing validator address would typically mean that the attacker doesn't own private key of the victim validator. This would cause loss of funds for the attacker, as the validator needs the private key corresponding to its address to retrieve the funds via redeeum or transferUnstakedOut etc.

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid: there is no function for that unstaking if its more than 300 in the array; medium(1)

noslav

partially fixed by
ignore 0 amount unstakings and require unstaking length < 300 - sa46,

nevillehuang

@sudeepdino008 @noslav

- What is the purpose of setValidatorAddress()? Is it simply to a feature to allow current validator to operate with another address (given they would have to call themselves, so I believe this does not cater to a private key loss scenario)
- Can the validator still perform regular functionalities such as redeeming rewards, commissions submit proofs with no issue?

If above scenarios are true, I believe this issue is low severity

noslav

> @sudeepdino008 @noslav
>
> - What is the purpose of setValidatorAddress()? Is it simply to a feature to allow current validator to operate with another address (given they would have to call themselves, so I believe this does not cater to a private key loss scenario)
> - Can the validator still perform regular functionalities such as redeeming rewards, commissions submit proofs with no issue?
>
> If above scenarios are true, I believe this issue is low severity

@nevillehuang This is indeed true, the function does not cater to the private key loss scenario but rather a private key leak scenario where an unknown entity has taken hold of the private key, the action required is for the original owner entity to quickly switch the validator staking address to another address they control to thereby save the stake also from being stolen. This function now has the check that it's not another existing address in the system that belongs to a delegator or a validator.

In the case of a complete loss of private key, the validator is responsible for any such full loss and has to deal with the "not your keys not your crypto" essentially having no recourse to `_unstake` `_stake` or `_redeemRewards` with the same address

sherlock-admin

> Escalate
>
> this issue should be Invalid.
>
> This is already a known issue by the team, and was described in the readMe
>
> > When changing its address a validator cannot transfer unstakings if there are more than 300 of them. This is to ensure the contract does not revert from too much gas used. In case if there are more than 300 unstakings, there is an option to transfer the address without unstakings.
>
> https://github.com/sherlock-audit/2023-11-covalent/blob/main/cqt-staking/README.md
>
> scroll down to the `Staking Explained` section to find it.

You've deleted an escalation for this issue.

midori-fuse

I disagree with the escalation.

The first part (cannot transfer unstakings) is known, but the "In case if there are more than 300 unstakings..." part is not. Report #67 proves that it is not possible even for an admin to transfer the validator address without unstakings, so the quoted README part actually confirms that a core contract functionality is broken.

ArnieGod

@midori-fuse i agree with you removing escalation.

ArnieGod

@midori-fuse escalation removed thanks for pointing out my misjudgment.

CergyK

Fix LGTM

MLON33

The protocol team has acknowledged the issue.

Issue M-2: OperationalStaking may not possess enough CQT for the last withdrawal

Source: https://github.com/sherlock-audit/2023-11-covalent-judging/issues/39

Found by

aslanbek, bitsurfer, cheatcode, dany.armstrong90

Summary

Both `_sharesToTokens` and `_tokensToShares` round down instead of rounding off against the user. This can result in users withdrawing few weis more than they should, which in turn would make the last CQT transfer from the contract revert due to insufficient balance.

Vulnerability Detail

1. When users `stake`, the shares they will receive is calculated via `_tokensToShares`:

```
function _tokensToShares(
    uint128 amount,
    uint128 rate
) internal view returns (uint128) {
    return uint128((uint256(amount) * DIVIDER) / uint256(rate));
}
```

So the rounding will be against the user, or zero if the user provided the right amount of CQT.

2. When users unstake, their shares are decreased by

```
function _sharesToTokens(
    uint128 sharesN,
    uint128 rate
) internal view returns (uint128) {
    return uint128((uint256(sharesN) * uint256(rate)) / DIVIDER);
}
```

So it is possible to `stake` and `unstake` such amounts, that would leave dust amount of shares on user's balance after their full withdrawal. However, dust amounts can not be withdrawn due to the check in _redeemRewards:

```
require(
    effectiveAmount >= REWARD_REDEEM_THRESHOLD,
    "Requested amount must be higher than redeem threshold"
);
```

But, if the user does not withdraw immediately, but instead does it after the multiplier is increased, the dust he received from rounding error becomes withdrawable, because his `totalUnlockedValue` becomes greater than `REWARD_REDEEM_THRESHOLD`.

So the user will end up withdrawing more than their `initialStake + shareOfRewards`, which means, if the rounding after all other operations stays net-zero for the protocol, there won't be enough CQT for the last CQT withdrawal (be it `transferUnstakedOut`, `redeemRewards`, or `redeemCommission`).

Foundry PoC

Impact

Victim's transactions will keep reverting unless they figure out that they need to decrease their withdrawal amount.

Code Snippet

https://github.com/sherlock-audit/2023-11-covalent/blob/main/cqt-staking/contracts/OperationalStaking.sol#L386-L388

https://github.com/sherlock-audit/2023-11-covalent/blob/main/cqt-staking/contracts/OperationalStaking.sol#L393-L395

Tool used

Manual Review

Recommendation

`_sharesToTokens` and `_tokensToShares`, instead of rounding down, should always round off against the user.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid: watson explained how rounding error would prevent the the last to withdraw the chance to unless there are some changes in place; medium(5)

noslav

The issue lies in this check

```
require(
    effectiveAmount >= REWARD_REDEEM_THRESHOLD,
    "Requested amount must be higher than redeem threshold"
);
```

where the value by default for REWARD_REDEEM_THRESHOLD is 10*8 and hence redemption below that value is not possible leading to the build up of dust as the issue describes until that threshold is crossed

noslav

fixed by round up sharesToBurn and sharesToRemove due to uint258 to uint128 co

rogarcia

correct PR commit https://github.com/covalenthq/cqt-staking/pull/125/commits/5a771c3aa5f046c06bd531f0f49530fb7d7bfdee

CergyK

Fix LGTM

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/covalenthq/cqt-staking/pull/125/commits/1f957c05aacfb765d751a5ec3cbfd1798e1fae15.

sherlock-admin

The Lead Senior Watson signed off on the fix.

Issue M-3: New staking between reward epochs will dilute rewards for existing stakers. Anyone can then front-run `OperationalStaking.rewardValidators()` to steal rewards

Source: https://github.com/sherlock-audit/2023-11-covalent-judging/issues/47

Found by

Atharv, Bauer, PUSH0, alexbabits, aslanbek, cawfree, cergyk, hunter_w3b, ljj, petro1912, thank_you

Summary

In the Covalent Network, validators perform work to earn rewards, which is distributed through `OperationalStaking`. The staking manager is expected to regularly invoke `rewardValidators()` to distribute staking rewards accordingly to the validator, and its delegators.

However, the function takes into account all existing stakes, including new ones. This makes newer stakes being counted equally to existing stakes, despite newer stakes haven't existed for a working epoch yet.

An attacker can also then front-run `rewardValidators()` to steal a share of the rewards. The attacker gains a share of the reward, despite not having fully staked for the corresponding epoch.

Vulnerability Detail

The function `rewardValidators()` is callable only by the staking manager to distribute rewards to validators. The rewards is then immediately distributed to the validator, and all their delegators, proportional to the amount staked.

However, any new staking in-between reward epochs still counts as staking. They receive the full reward amount for the epoch, despite not having staked for the full epoch.

An attacker can abuse this by front-run the staking manager's distribution with a stake transaction. The attacker stakes a certain amount right before the staking manager distributes rewards, then the attacker is already considered to have a share of the reward, despite not having staked during the epoch they were entitled to.

This also applies to re-stakings, i.e. unstaked tokens that are re-staked into the same validator: Any stake recovers made through `recoverUnstaking()` is

considered a new stake. Therefore an attacker can use the same funds to repeatedly perform the attack.

Proof of concept

0. Alice is a validator. She has two delegators:
    - Alicia delegating 5000 CQT.
    - Alisson delegating 15000 CQT.
    - Alice herself stakes 35000 CQT, for a total of 50000.
1. The staking manager distributes 1000 CQT to Alice. This is then distributed proportionally across Alice and her delegators.
2. Bob notices the staking manager, and front-runs with a 50000 CQT stake into Alice.
    - Alice now has a total stake of 100000, with half of it belonging to Bob.
    - Bob's staking tx goes through *before* the staking manager's tx.
3. Bob now owns half of Alice's shares. He is distributed half of the rewards, despite having staked into Alice for only one second.
4. Bob can repeat this attack for as often as they wish to, by unstaking then restaking through `recoverUnstaking()`.

Impact

Unfair distribution of rewards. New stakers get full rewards for epochs they didn't fully stake into.

Code Snippet

https://github.com/sherlock-audit/2023-11-covalent/blob/main/cqt-staking/contracts/OperationalStaking.sol#L262

Tool used

Manual Review

Recommendation

Use a checkpoint-based shares computation. An idea can be as follow:

- For new stakings, don't mint shares for them right away, but add them into a "pending stake" variable.

- For unstakings, do burn their shares right away, as with the normal setting.
- When reward is distributed, distribute to existing stakes first (i.e. increase the share price only for the existing shares), only then mint new shares for the pending stakes.

Discussion

sudeepdino008

- there is a bit of unpredictability with which rewardValidators is called. It might not be possible to ascertain when it will be called.
- What value does Bob get by getting in just before the reward distribution, and then getting out afterwards? Why would he not simply remain staked?
- Note that for each quorum, the stake states (needed for reward calculation) is fetched for the block height (in which that particular quorum was emitted). Even though rewardValidators is called every 12 hours, the offchain system is doing a lot more frequent stake state checks.

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid: users that stakes immediately will be accounted for during the reward distribution; medium as there is no loss of funds; medium(4)

nevillehuang

1. This can be done by observing the mempool for calls to `rewardValidators()` given staking contract is deployed on mainnet
2. I believe the PoC in #34 can help you understand a possible impact:

   In the simple demonstration below, we show that an attacker _BOB may steal ~66% of Validator _ALICE's rewards by frontrunning an incoming call to [rewardValidators(uint128,uint128],uint128[]):
3. How does this prevent rewards siphoning stated in this issue?

sudeepdino008

1. you can observe the mempool for rewardValidators, but by that point, the offchain system has already calculated the validator rewards by taking the stake states into account i.e. when the call is made for rewardValidators, the rewards for each validator is already decided. Front running this call wouldn't change anything.

2. a cooldown for recoverUnstaking is implemented which prevents delegators from entering and exiting staked positions with the same validator.
3. BOB really doesn't "steal" the rewards; He entered the staking position at a particular time and was part of the staking pool. Surely we can make it difficult for them to re-enter after exiting, which is what 2. achieves.

https://github.com/covalenthq/cqt-staking/pull/125/commits/a609cca0426cb22cbf5064212341c14c288efeda for 2.

CergyK

Fix LGTM

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/covalenthq/cqt-staking/commit/a609cca0426cb22cbf5064212341c14c288efeda.

sherlock-admin

The Lead Senior Watson signed off on the fix.

Issue M-4: Frontrunning validator freeze to withdraw tokens

Source: https://github.com/sherlock-audit/2023-11-covalent-judging/issues/50

Found by

PUSH0

Summary

Covalent implements a freeze mechanism to disable malicious Validators, this allows the protocol to block all interactions with a validator when he behaves maliciously. Covalent also implements a timelock to ensure tokens are only withdraw after a certain amount of time. After the cooldown ends, tokens can always be withdrawn.

Following problem arise now: because the tokens can always be withdrawn, a malicious Validator can listen for a potential "freeze" transaction in the mempool, front run this transaction to unstake his tokens and withdraw them after the cooldown end.

Vulnerability Detail

Almost every action on the Operational Staking contract checks if the validator is frozen or not:

```
require(!v.frozen, "Validator is frozen");
```

The methods transferUnstakedOut() and recoverUnstaking() are both not checking for this, making the unstake transaction front runnable. Here are the only checks of transferUnstakedOut():

```
require(validatorId < validatorsN, "Invalid validator");
        require(_validators[validatorId].unstakings[msg.sender].length >
↪   unstakingId, "Unstaking does not exist");
        Unstaking storage us =
↪   _validators[validatorId].unstakings[msg.sender][unstakingId];
        require(us.amount >= amount, "Unstaking has less tokens");
```

https://github.com/sherlock-audit/2023-11-covalent/blob/main/cqt-staking/contracts/OperationalStaking.sol#L559-L572

This makes following attack possible:

1. Validator cheats and gets rewarded fees.
2. Protocol notices the misbehavior and initiates a Freeze transaction
3. Validator sees the transaction and starts a unstake() transaction with higher gas.
4. Validator gets frozen, but the unstaking is already done
5. Validator waits for cooldown and withdraws tokens.

Now the validator has gained unfairly obtained tokens and withdrawn his stake.

Impact

Malicious validators can front run freeze to withdraw tokens.

Code Snippet

https://github.com/sherlock-audit/2023-11-covalent/blob/main/cqt-staking/contracts/OperationalStaking.sol#L559-L572

Tool used

Manual Review

Recommendation

Implement a check if validator is frozen on `transferUnstakedOut()` and `recoverUnstaking()`, and revert transaction if true.

If freezing all unstakings is undesirable (e.g. not freezing honest unstakes), the sponsor may consider storing the unstake timestamp as well:

- Store the unstaking block number for each unstake.
- Freeze the validator from a certain past block only, only unstakings that occur from that block onwards will get frozen.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> invalid: this is theoretically not possile due to the cooldown time; the time will allow the governance to pause the contract/function

noslav

fixed by check validator not frozen for recoverUnstaking transferUnstakedOut

nevillehuang

Invalid, both are user facing functions, not validators.

Oot2k

Escalate

I believe this issue was mistakenly excluded, the frontrunning of freeze transaction is indeed a problem like described in the Report.

The impact described is clearly medium, because this attack makes the freeze function almost useless. Also it generates clear loss of funds for the protocol, because in most cases a malicious validator might accrue rewards which do not belong to him.

This issue can not really be fixed by governance pausing the contract, this would pause the contract for everyone else aswell and cause even more damage.

The fix by protocol teams looks good.

To summarize: Issue is fixed, impact is High, issue should be open and valid.

sherlock-admin

> Escalate
>
> I believe this issue was mistakenly excluded, the frontrunning of freeze transaction is indeed a problem like described in the Report.
>
> The impact described is clearly medium, because this attack makes the freeze function almost useless. Also it generates clear loss of funds for the protocol, because in most cases a malicious validator might accrue rewards which do not belong to him.
>
> This issue can not really be fixed by governance pausing the contract, this would pause the contract for everyone else aswell and cause even more damage.
>
> The fix by protocol teams looks good.
>
> To summarize: Issue is fixed, impact is High, issue should be open and valid.

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.

midori-fuse

Adding to the escalation point, there is no way for governance to forcefully claim an unstaking (or any rewards that has been distributed). Therefore eventually the contract must be unpaused to avoid locking of existing funds, and the malicious actor fully gets away.

Oot2k

Additionally I think this issue should be judged as HIGH severity based on following facts:

- it create a clear loss of funds for the protocol (The main reason to freeze a validator is to penalize him for malicious behavior, this can include stealing funds / rewards)
- there is no way to prevent this behavior without causing more damage
- attack cost is really low -> just transaction fee

nevillehuang

@Oot2k @midori-fuse @noslav Could you shed more details on how the validator can cheat and get rewarded fees and in what scenarios is a freeze initiated. It seems to me like a hypothetical scenario given my understanding is validator is still unstaking amounts that belongs to him, but could be significant.

1. Validator cheats and gets rewarded fees.

Additionally, this issue seems to be dependent on a front-running attack vector, so:

- If flashbots are considered similar to issue here to mitigate front-running, I believe this could be low severity
- If not, if it is true that the freeze mechanism can be bypassed, then I believe this is medium severity, since it is dependent on a hypothetical scenario that validators turn malicious.

Additionally, is there any mechanisms in place to mitigate malicious validators?

midori-fuse

Providing evidence for the bypassing of freeze mechanism. Search the following phrase within the contract:

```
require(!v.frozen, "Validator is frozen");
```

It appears 6 times throughout the contract, and covers all entry points except `transferUnstakedOut()` (except admin and reward manager functions). Eyeballing all other external functions (except the ones mentioned) will show that they all go through `_stake()` or `_unstake()`, which has the appropriate check.

For transferUnstakedOut(), there is no check for whether the validator corresponding to that unstake has been frozen or not, neither is there in _transferFromContract().

The flow for an unstaking to happen (for delegators or validators alike) is that:

- The user calls `unstake()` or `unstakeAll()`.
- Wait for the cooldown.
- Call `transferUnstakedOut()` to actually receive those tokens.

Then the freeze is bypassed if the user is able to call unstake before the validator is frozen. Front-running is only required to maximize the getaway amounts by squeezing some extra rewards, you can just unstake before getting freeze and you bypass the freeze already. Therefore this is just a bypassing of freeze, and not dependent on front-running. We simply show the scenario which has the maximum impact.

midori-fuse

For the scenario where a validator cheats, there are certain ways for it to happen:

- Two or more validators collude and are able to force quorum on certain sessions, earning them rewards on an incorrect block specimen.
- A validator finds a systemic exploit and/or simply not doing work correctly (e.g. admin determined them to repeatedly copy-paste other validators' works by watching the mempool or copying existing submissions, despite it being wrong or not). Note that a disabled validator can still unstake and get away with funds, unlike the frozen scenario (without the current bypassing issue).

The freeze is there to protect against these kind of situations.

nevillehuang

I personally am not convinced of this issue because the admins can always perform a system wide contract pause before freezing validators in separate transactions via flashbots (which pauses all actions, including `transferUnstakedOut`), which possibly mitigates this issue. This is in addition to the fact that there is a 28 day unstaking cooldown period which is more than sufficient time to react to malicious validators by admins (which in itself is a mitigation).

So I will leave it up to @Czar102 and sponsors to decide validity.

midori-fuse

28 day unstaking cooldown does not mitigate this. As soon as the unstaking is done, the amount can be transferred out after 28 days (and the admin unpauses the system). Even if the system is paused, there are no admin actions that can revoke an unstaking that's on cooldown.

Keeping the system paused equates to locking all funds, including other validators' funds and their respective delegators, and the admin still cannot take over the stolen funds.

Furthermore the issue shows that freezing can be bypassed, and front-running is not a condition. The validator can just unstake right after the exploit, and the admin is already powerless before noticing the issue.

Just adding some extra points. As part of the team making the escalation, we have the responsibility to provide extra information and any context the judges' might have missed, but we respect the judges' decision in any case.

Oot2k

Agree with @midori-fuse here. Cooldown -> does not do anything because the malicious user still transfers tokens out (this is the root cause of this issue) Admin can pause protocol -> this will pause all actions for other users as well, as soon as the protocol is unpaused funds can be withdrawn again Malicious funds -> yes this report assumes there is a way to get funds maliciously and for this reason the team implemented the freeze mechanism

I think this summarizes the issue pretty well and should be enough for Sherlock to validate.

nevillehuang

If the hypothetical scenario of a way to cheat funds/validators being malicious is considered as a valid reason that break admin initiated pause mechanism, I can agree this is a valid medium since I believe the only way to resolve the issue is for the owner to perform an upgrade to the contract.

Although I must say, the whole original submission is only presenting a front-running scenario, and the watsons only realized after that front-running is not necessary but did not include it in the original submission, and hence my arguments.

Czar102

Great points made, the frontrunning argument is also not convincing to me since it's quite clear that this race condition is by design and it's admin's responsibility to keep the information about the freeze private until confirmation.

But, this issue can also be considered a loss of functionality (freezing stakes) due to the existence of a beneficial optimal game-theoretic behavior of the attacker.

I'm currently inclined to accept this as a Medium severity issue.

Czar102

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- Oot2k: accepted

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/covalenthq/cqt-staking/pull/125/commits/de86308999d093a3f4553aa7094ed4d29be8beb0.

CergyK

Fix LGTM

sherlock-admin

The Lead Senior Watson signed off on the fix.

Issue M-5: No cooldown in `recoverUnstaking()`, opens up several possible attacks by abusing this functionality.

Source: https://github.com/sherlock-audit/2023-11-covalent-judging/issues/52

Found by

PUSH0

Summary

The function `recoverUnstaking()` allows a user to recover any of their unstaking to the same validator.

However, the function has no cooldown, i.e. a user can unstake and restake to the same validator at any time they prefer.

We describe several attack vectors that may arise from this issue.

Vulnerability Detail

When a user unstakes, they must wait for a cooldown of 28 days (180 days if validator) before being able to withdraw. The user also has the choice to revert the action, and re-stake said amount back to the same validator by calling `recoverUnstaking()`.

However, the function `recoverUnstaking()` has no cooldowns. In other words, a user can re-stake as many times as they like, at any time they prefer, without being subject to the unstaking cooldown.

This opens up several possible attack vectors by abusing the lack of cooldown of this functionality.

Next section we describe some of the possible attack vectors.

Denying delegators for any validators

An attacker with a large enough capital can do the following to deny a target validator from getting delegated stakings:

- Max stake to the target validator, up to `staked * maxCapMultiplier`, but immediately unstake.
- Whenever someone stakes, front-run them with a `recoverUnstaking()`, re-staking their max stake.

- The user's stake tx fails, as the attacker has taken up the maximum possible stake.
- The attacker then unstakes, able to repeat the attack when necessary.

The impact is that a validator can be denied from getting delegators.

Stealing of rewards

This attack vector builds upon a separate issue we have submitted, about unfair distribution of rewards for mid-epoch stakers.

An attacker can perform the following, while greatly de-risking themselves from staking.

- Stake an amount they want, but unstake.
- Listen to the staking manager's `rewardValidators()` with the intent of front-running.
- If the attacker choose to, front-run `rewardValidators()` calls, unfairly taking a share of the rewards.
- Unstake.

The attacker benefits from the following:

- While unstaking, their 28-day cooldown still counts. They greatly de-risk themselves by being able to choose whether or not to steal rewards (and reset countdown), or to take profits.
- Unstakings from frozen validators can still be withdrawn once the cooldown has passed. By keeping themselves in an unstaking state, they bypass the freezing mechanism.
- They eliminate the risk from other standard staking mechanism, should they be implemented in the future (e.g. slashing).

Impact

`recoverUnstaking()` can be abused, opens up several possible attack vectors, with varying impacts (shown above).

Code Snippet

https://github.com/sherlock-audit/2023-11-covalent/blob/main/cqt-staking/contracts/OperationalStaking.sol#L542

Tool used

Manual Review

Recommendation

Implement a cooldown for `recoverUnstaking()`. A short cooldown (e.g. a few days) is sufficient.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> invalid: the coolDown is in place for withdrawing tokens not for reStaking 'em

noslav

fixed by
implement recoverUnstakingCoolDown period for recoverUnstaking levera...

noslav

supporting fixes by https://github.com/covalenthq/cqt-staking/commit/a609cca0426cb22cbf5064212341c14c288efeda

CergyK

Fix LGTM

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/covalenthq/cqt-staking/pull/125/commits/e7ab9ab7eb89f47669dfc0c4ef175f6ca074328b.

sherlock-admin

The Lead Senior Watson signed off on the fix.

Issue M-6: `validatorMaxStake` can be bypassed by using `setValidatorAddress()`

Source: https://github.com/sherlock-audit/2023-11-covalent-judging/issues/66

Found by

Al-Qa-qa, PUSH0, SadBase, aslanbek, cergyk, petro1912, qmdddd, zach223

Summary

`setValidatorAddress()` allows a validator to migrate to a new address of their choice. However, the current logic only stacks up the old address' stake to the new one, never checking `validatorMaxStake`.

Vulnerability Detail

The current logic for `setValidatorAddress()` is as follow:

```
function setValidatorAddress(uint128 validatorId, address newAddress) external
 ↪  whenNotPaused {
    // ...
    v.stakings[newAddress].shares += v.stakings[msg.sender].shares;
    v.stakings[newAddress].staked += v.stakings[msg.sender].staked;
    delete v.stakings[msg.sender];
    // ...
}
```

The old address' stake is simply stacked on top of the new address' stake. There are no other checks for this amount, even though the new address may already have contained a stake.

Then the combined total of the two stakings may exceed `validatorMaxStake`. This accordingly allows the new (validator) staker's amount to bypass said threshold, breaking an important invariant of the protocol.

Proof of concept

0. Bob the validator has a self-stake equal to `validatorMaxStake`.
1. Bob has another address, B2, with some stake delegated to Bob's validator.
2. Bob migrates to B2.
3. Bob's stake is stacked on top of B2. B2 becomes the new validator address, but their stake has exceeded `validatorMaxStake`.

4. B2 can then repeated this procedure to addresses B3, B4, ..., despite B2 already holding more than the max allowed amount.

Bob now holds more stake than he should be able to, allowing him to earn an unfair amount of rewards compared to other validators.

We also note that, even if the admin tries to freeze Bob, he can front-run the freeze with an unstake, since unstakes are not blocked from withdrawing (after cooldown ends).

Impact

- Breaking an important invariant of the protocol.
- Allowing any validator to bypass the max stake amount. In turn allows them to earn an unfair amount of validator rewards in the process.
- Allows a validator to unfairly increase their max delegator amount, as an effect of increasing `(validator stake) * maxCapMultiplier`.

Code Snippet

https://github.com/sherlock-audit/2023-11-covalent/blob/main/cqt-staking/contracts/OperationalStaking.sol#L689-L711

Tool used

Manual Review

Recommendation

Check that the new address's total stake does not exceed `validatorMaxStake` before proceeding with the migration.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid: user can exceed max deposit and potentially increase his max delegator amoun; high(1)

noslav

fixed by prevent new validator address stake from exceeding max stake - sa66/s...

CergyK

Fix LGTM

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/covalenthq/cqt-staking/pull/125/commits/0eba6b318b9400e4a5d6511ba4c96922b83b9abd.

sherlock-admin

The Lead Senior Watson signed off on the fix.

Issue M-7: OperationalStaking::_unstake Delegators can bypass 28 days unstaking cooldown when enough rewards have accumulated

Source: https://github.com/sherlock-audit/2023-11-covalent-judging/issues/78

Found by

cergyk, irresponsible, qmdddd

Summary

When rewards are distributed, they are distributed evenly accross all of the shares held for a validator. However participants collecting rewards burn shares, and thus receive less rewards in subsequent rounds, compared to participants which have left rewards in the contract. This means that delegators can unstake instead of redeeming rewards, which will progressively replace the initial staked amount with claimable rewards.

Vulnerability Detail

We can see in the function `redeemRewards` that the equivalent amount of shares is burned to redeem: https://github.com/sherlock-audit/2023-11-covalent/blob/main/cqt-staking/contracts/OperationalStaking.sol#L616-L631

This means that by redeeming rewards a validator will receive less of the future rewards than the stake he invested.

Conversely, this means that delegators can progressively derisk their staking position by burning small quantity of shares from their stake (by calling `unstake`) instead of claiming rewards. When enough rewards accumulated, they have the equivalent of their initial amount in the contract, but it is less risky, since as rewards, they are not subject to the unstaking cooldown of 28 days and can be transferred out immediately.

Scenario

Alice is a validator staking 35 000 CQT. Bob delegates 35 000 CQT to Alice.

Shares distribution:

- Alice owns 35 000e18 shares
- Bob owns 35 000e18 shares

Owner distributes rewards to the tune of 70000 CQT (an exaggerated amount to make the point here)

Alice withdraws all her rewards, and burning half of her shares Bob unstakes all of his position, but does not redeem his rewards

Shares distribution:

- Alice owns 17 500e18 shares
- Bob owns 17 500e18 shares

Alice and Bob still have the same number of shares, and have the same claim to future rewards. However Bob's position has less risk compared to a normal delegator, since he can get all of his funds out anytime by calling `redeemRewards`, and does not have to wait a cooldown period.

Impact

Code Snippet

Tool used

Manual Review

Recommendation

Use a `claimedRewards` mapping to track already claimed rewards instead of burning shares when redeeming rewards

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> invalid: should provide a POC

noslav

fixed partially by
prevent bypassing cooldown with redelegateUnstaked - sa78/sa68/sa76

rogarcia

PR: https://github.com/covalenthq/cqt-staking/pull/125 commit: https://github.com/covalenthq/cqt-staking/pull/125/commits/5bf8940c8d5642652b1987cc74cb2f6780b06b08

sherlock-admin

> Escalate
>
> issue #68 is not a duplicate of this issue at all and should be its own issue.

You've deleted an escalation for this issue.

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/covalenthq/cqt-staking/pull/125/commits/5bf8940c8d5642652b1987cc74cb2f6780b06b08.

sherlock-admin

The Lead Senior Watson signed off on the fix.

Issue M-8: BlockSpecimenProofChain::submitBlockSpecimenPro Block specimen producer can greatly reduce session duration by submitting fake block specimen in the future

Source: https://github.com/sherlock-audit/2023-11-covalent-judging/issues/79

Found by

cergyk

Summary

Block specimen producers submit specimens for a given block number during a limited time called a session. Any block producer can start a session by calling `submitBlockSpecimenProof` for a given block number. This means that a block specimen producer can start a session for a block which does not exist yet, and severily reduce the actual session time, since honest block specimen producer can only participate between the time the block has been created and the end of session.

Vulnerability Detail

We can see that a session is started when the first specimen for the block height is submitted: https://github.com/sherlock-audit/2023-11-covalent/blob/main/cqt-staking/contracts/BlockSpecimenProofChain.sol#L346-L348

This means that if a malicious block specimen producer has sent some invalid data for a block height which is in the future, the session is still started for that block. The following check ensures that a producer can not call submit for a block too far in the future: https://github.com/sherlock-audit/2023-11-covalent/blob/main/cqt-staking/contracts/BlockSpecimenProofChain.sol#L344

But since the default value for `cd.allowedThreshold` would be 100 blocks, and a session duration would be approximately 240 blocks, we can see that a malicious block producer can reduce the actual session duration for honest producers by half.

Impact

The block specimen production session can be greatly reduced by a malicious producer (up to a half with current deploy parameters).

Code Snippet

Tool used

Manual Review

Recommendation

Please consider starting the session at the estimated timestamp of the considered `blockHeight`:

```
- session.sessionDeadline = uint64(block.number + _blockSpecimenSessionDuration);
+ uint64 timestampOnDestChain = (blockHeight-cd.blockOnTargetChain)*cd.secondsPe
↪  rBlock-cd.blockOnCurrentChain*_secondsPerBlock;
+ session.sessionDeadline = uint64(timestampOnDestChain +
↪  _blockSpecimenSessionDuration);
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid: user can submit fake here also; medium(3)

noslav

we can mitigate this a bit although not completely as we're currently tied to this way of doing things. We use the block numbers on the current chain (aka current block number on moonbeam + number of blocks to wait for session duration) to determine the deadline for any input block specimen number.. there's currently no way to know the head of the source chain (ethereum) except through the estimated calculation being done in the contract where block time * time diff from last chain sync tx and we can use to create a shorted upper bound so blocks too far in the future cannot be submitted!

noslav

partially fixed by
impl proof submission upper bounds to mitigate future block deadlines

CergyK

Fix LGTM

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/covalenthq/cqt-staking/pull/125/commits/481fcd4ea97e7f6e998dd30ef15122a8e256e5dc.

sherlock-admin

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.