SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Public
Prepared for:	ALEO
Prepared by:	Sherlock
Lead Security Expert:	dtheo
Dates Audited:	June 10 - June 24, 2024
Prepared on:	September 19, 2024

Introduction

An all-in-one ZK platform for web applications that are programmable, permissionless, and secure.

Scope

Repository: AleoNet/snarkVM

Branch: mainnet

Commit: 74a437897e80514cb88e52a3033df5f69181aed8

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
2	2

Security experts who found valid issues

dtheo sammy

Issue H-1: Aleo prover/network DOS vector due to invalid `split` proofs being free to abuse

Source: https://github.com/sherlock-audit/2024-05-aleo-judging/issues/34

Found by

dtheo

Summary

Split transactions that contain only 1 transition (the entire transaction is a single `split` transaction and nothing else) do not require a fee commitment. This allows for a malicious entity to grieve provers by forcing them to attempt and verify invalid proofs with no monetary loss. This can be used by competing provers to DOS their competition and waste CPU cycles that would otherwise be used towards the "mining/proving" the next block. A malicious actor attack multiple provers can significantly slow or stop aleo blocks from being created.

Vulnerability Detail

Aleo has `fee`, `deploy` and `execute` transaction types. If a `deploy` or an `execute` transaction type is broadcasted with an invalid proof or if the `finalize` fails then their corresponding `fee` transaction will be consumed instead. This is a DOS prevention feature that Aleo uses to prevent the network from having to verify proofs without some amount of collateral fee on the side.

Single split transactions, when called by users directly, are a special case where th e there is no fee commitment required. Aleo attempts to charge the fee directly in the split function by burning it. The issue is that if there is not sufficient funds to pay the fee and the proof fails then their is no backup fee commitment to seize. This opens up the Aleo network to a DOS vector that that is nearly free to abuse.

An attacker must only pay the initial fees to make a large number of private records with at least 1 microcredit each. As long as they only use each credit once per block , they can reuse the records to to make invalid proofs that will pass all of the transaction prechecks in `check_fee`, `check_transaction`, etc. until the proof verification is attempted and CPU cycles are wasted.

Impact

This can be used to DOS Aleo provers by other provers competing to solve the Aleo Coinbase Puzzle (and gain rewards). If a malicious entity attacked all provers at the same time the Aleo network would stop producing blocks during this time.

Code Snippet

snarkVM/synthesizer/src/vm/verify.rs#L225-L248

snarkVM/synthesizer/program/src/resources/credits.aleo#L947-L972)

Tool used

Manual Review

Recommendation

Require a fee commitment outside of the split transaction than can be verified quickly in the transaction precheck/speculation flow.

Discussion

sammy-tm

@evanmarshall

Can you tell us your thoughts about this issue?

Issue H-2: `split` transaction's fixed fees undercharge block stuff DOS attacks

Source: https://github.com/sherlock-audit/2024-05-aleo-judging/issues/35

Found by

dtheo

Summary

Split transactions that contain only 1 transition (the entire transaction is a single `split` transaction and nothing else) do not require a fee commitment and instead charge a fixed 10 credit fee. Since the fee is fixed this transaction type can be abused in block stuffing attacks without having to increase the fee during times of congestion.

Vulnerability Detail

Similar to Ethereum's EIP-1559 mechanism, Aleo allows for fee increases during times of congestion to prevent DOS due to block stuffing attacks. Unlike public and private `fee` transaction types where the fee is dynamic and provided as inputs, `split` transaction types have a fixed fee that is hardcoded to 10 credits in the `split` function in credits.aleo. This allows for an attacker to use this transaction type to fill blocks to their max transaction limit without ever having to increase their fees. This allows for unreasonable cheap block stuffing attacks that can be used to DOS the network during critical time periods to prevent DEX liquidations, collateral adjustments, etc.

Impact

The hardcoding of the `split` transactions fee amount and lack of outer fee commitment requirements bypass the Aleo networks dynamic fee mechanism allowing unreasonable cheap block stuff DOS attacks on the network.

Code Snippet

snarkVM/synthesizer/src/vm/verify.rs#L225-L248

snarkVM/synthesizer/program/src/resources/credits.aleo#L947-L972)

snarkVM/ledger/block/src/verify.rs#L422-L428

Tool used

Manual Review

Recommendation

Remove the fixed fee from inside the `split` function in credits.aleo. Require fee commitments for this transaction type that are similar to all other transaction types.

Discussion

evanmarshall

Somewhat a duplicate of: https://github.com/sherlock-audit/2024-05-aleo-judging/issues/17

A fee is already charged (at the program layer instead of consensus layer). The base fee is about twice was a `transfer_private` requires as a base fee.

sherlock-admin3

> Escalate
>
> The sponsors comment indicates that the split fee is 2 times the transfer_private fee so undercharging is not occurring here.

You've deleted an escalation for this issue.

infosecual

> The sponsors comment indicates that the split fee is 2 times the transfer_private fee so undercharging is not occurring here.

The initial cost of the transaction at baseline is 2x. This submission is not talking about the baseline cost. This submission is talking about the fact that the split transaction costs will not increase with every other transaction type when congestion is happening. Aleo Documentation on transaction fees explains this a bit:

> The minimum execute transaction fee may increase to prevent spamming; it is currently stable between 3 and 5 millicredits for a basic credits.aleo transfer.

For example, `transfer_private` fees will increase and could easily 20-50x during periods of spam/congestion- think similar to an evm chain at 150 gwei gas prices. In Aleo the `split` fee is waived at the consensus layer and is exactly 10 credits at the program layer. Since all dynamic fee pricing happens at the consensus layer the `split` transaction will never increase in cost and can be used to spam without increasing costs. This can crowd out other transactions during congestion even as all other `execute` transaction get multiple magnitudes more expensive.

evanmarshall

So this isn't implemented yet in the snarkOS repo but I'd imagine that validators will drop transactions that generate the least revenue in the long run. I don't see it as a spam vector as validators noticing a full mempool will simply prioritize the most profitable transactions.

A bigger criticism that I don't believe the mempool is currently sorted by priority fees. Once that is in place, I see the issue as more DOS for the split transaction. If validators are sorting the mempool by profitability and fees are high, no split transaction will ever get through because it won't be profitable with a fixed fee.

Haxatron

> The initial cost of the transaction at baseline is 2x. This submission is not talking about the baseline cost. This submission is talking about the fact that the split transaction costs will not increase with every other transaction type when congestion is happening.

Apologies, I stand corrected. I do agree that the split fee not being subjected to EIP-1559 style pricing is a real issue and will delete the escalation.

sherlock-admin3

> Escalate
>
> This is valid. However, I would like to dispute the severity of this issue.
>
> It does not meet Sherlock's High Severity criteria :
>
> https://docs.sherlock.xyz/audits/judging/judging#iv.-how-to-identify-a-high-issue
>
> I believe it should be downgraded to a Medium instead

You've deleted an escalation for this issue.

infosecual

I am very confident of the severity of this issue.

The issue here is that there is currently no dynamic fee mechanism for the `split` transaction type and there is currently no logic in the sequencer to order transactions by priority fees, so there is nothing to prevent crowding out valid transactions with spam that is not subject to dynamic increasing costs. There are really two things that need to happen here to mitigate this issue - 1. a priority fee auction market ordering needs to be implemented and 2. a dynamic fee needs to be implemented for `split` in order to have its pricing adjust correctly during times of congestion.

To comment on the impact- if this is not fixed someone will eventually abuse this to destroy a defi protocol. Users and protocols built on Aleo will experience an attack like this as a chain liveness issue. Any protocol that depends on liquidations happening in a timely manner will be affected by this. Both users and protocols will incur material loss. Users will not be able to withdrawal assets or top up collateral, DEX's will not be able to liquidate positions that are about to incur bad debt against the protocol, etc. The list can go on and on since this breaks chain liveness assumptions that all protocols are built on.

For some reference, in Ethereum world this would be detrimental to Maker, Euler, AAVE, Curve, etc. The largest protocols there are. A malicious entity can abuse issues like this one to prevent critical transactions from landing during times of volatility which can lead to entire protocols going insolvent. This was one of the early concerns with Maker on Ethereum - everyone was afraid of a the possibility of a DAI depeg if Maker could not liquidate large vaults in time due to a targeted DOS attack during a large ETH price drop (shanghai attacks were in everyone's recent memory). A DAI depegging is just one example of what could happen during a chain liveness issue. Luckily Ethereum has not had a liveness issue since the creation of these protocols. For everyones sake I hope it never does and I hope Aleo never has to experience one either.

Sherlock is highly geared towards smart contract contests and does not have judging rules specific to L1s so there is no callout to specifically categorize the severity of this category of bug as high. However here are the two possible criteria for High on Shrelock - both are still met:

> Definite loss of funds without (extensive) limitations of external conditions.
>
> Inflicts serious non-material losses (doesn't include contract simply not working).

While this is not the sort of issue you are used to seeing on a smart contract audit platform, this is the type of issue that underpins all smart contracts. My day job is to focus on L1 security research and attacks like these are the sort of nightmare situation that keeps my team up at night. The EF bug bounty would consider this type of issue in an Ethereum client as a `critical` category and award the max payout. This is a foundational issue that needs to be addressed in order to have a stable DEFI economy on Aleo. It does not matter how secure the smart contracts are that run on Aleo if attacks like this are possible.

Issue M-1: `delegated[]` state is not removed after it reaches zero, potentially leading to higher computational costs and DoS

Source: https://github.com/sherlock-audit/2024-05-aleo-judging/issues/25

Found by

sammy

Summary

`delegated[]` mapping stores the amount of credits delegated to a validator. When this value reaches zero, it is not removed from storage, which can lead to higher computational costs.

Vulnerability Detail

When a validator is removed, the `delegated[]` mapping is not removed from storage : credits.aleo#L625-L663

```
position remove_validator;
// {
    /* Committee */


    // Remove the validator from the committee.
    remove committee[r5.validator];


    /* Metadata */


    // Retrieve the current committee size.
    get
↪   metadata[aleo1qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq3ljyzc]
↪   into r42;
    // Decrement the committee size by one.
    sub r42 1u32 into r43;
    // Store the new committee size.
    set r43 into
↪   metadata[aleo1qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq3ljyzc];
```

```
    /* Delegated */


    // Decrease the delegated total by the bonded amount.
    sub r31 r30.microcredits into r44;
    // Store the new delegated total.
    set r44 into delegated[r5.validator];


    /* Bonded */


    // Remove the bonded state.
    remove bonded[r5.validator];


    /* Unbonding */


    // Retrieve or initialize the unbonding state.
    get.or_use unbonding[r5.validator] r4 into r45;
    // Increment the unbond amount by the full bonded amount.
    // Note: If the addition overflows, reject the transition.
    add r30.microcredits r45.microcredits into r46;
    // Construct the updated unbond state.
    cast r46 r3 into r47 as unbond_state;
    // Store the new unbonding state.
    set r47 into unbonding[r5.validator];
```

This can cause problems in other areas of the code, where the entire `delegated[]` mapping is iterated over. For example, committee.rs#L73-L78

```
let Some(microcredits) = delegated_map.iter().find_map(|(delegated_key,
↪   delegated_value)| {
                // Retrieve the delegated address.
                let delegated_address = match delegated_key {
                    Plaintext::Literal(Literal::Address(address), _) =>
↪   Some(address),
                    _ => None,
                };
                // Check if the address matches.
                match delegated_address == Some(address) {
                    // Extract the microcredits from the value.
                    true => match delegated_value {
```

```
↪  Value::Plaintext(Plaintext::Literal(Literal::U64(microcredits), _)) =>
↪  Some(**microcredits),
                         _ => None,
                    },
                    false => None,
                }
            }) else {
                bail!("Missing microcredits for committee member - {address}");
            };
```

A malicious user can activate the delegated mapping for several addresses by repeatedly bonding and unbonding. This will make certain computations expensive and may even lead to a DoS if the computation becomes too expensive to execute.

Impact

DoS

Code Snippet

Tool used

Manual Review

Recommendation

Whenever the `delegated[]` mapping becomes 0, remove it from storage

Discussion

sammy-tm

Escalate

I think this issue is valid because the cost to iterate the delegated mapping increases with each initialization. Delegated state should be removed if it touches 0. A malicious user can initialize as many delegated[] key-value pairs as they want by bonding and unbonding repeatedly.

sherlock-admin3

> Escalate
>
> I think this issue is valid because the cost to iterate the delegated mapping increases with each initialization. Delegated state should be

> removed if it touches 0. A malicious user can initialize as many delegated[] pairs as they want by bonding and unbonding constantly.

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.

cvetanovv

I invalidated this issue because it's unclear what the actual impact would be. @evanmarshall @infosecual Can you give your opinion?

WangSecurity

> A malicious user can activate the delegated mapping for several addresses by repeatedly bonding and unbonding. This will make certain computations expensive and may even lead to a DoS if the computation becomes too expensive to execute.

So it's at most 1-block DOS, correct? Secondly, how expensive the gas is at Aleo?

sammy-tm

No, once the delegated mapping is activated for an address, there is no way to remove it from storage. Also, all other mappings require a significant bond commitment to remain active and are removed from storage once they reach 0. Hence the impact is permanent.

sammy-tm

You can understand the transaction fees here

The problem with the delegated[] mapping is that anyone can add to it's storage as long as they have atleast 10_000 credits, they can bond and unbond to ANY address (even if it's not a real address in the network). The only cost to carry out the mentioned attack path is the gas fee. This will increase storage costs and can potentially slow down transactions/ increase fee, leading to DoS.

IWildSniperI

in what cases do we need to loop into that mapping?, is it in every txn? that will lead to chain halting? @sammy-tm

sammy-tm

Yes, the delegated map is iterated over in every single transaction in the network since it is used to calculate the staking rewards. Which is why it is a bigger concern than a normal smart contract having unbounded storage. Every transaction unrelated to credits.aleo will iterate over the mapping as well, so the impact is not just limited to credits.aleo.

sammy-tm

```
fn test_bond_delegator_to_multiple_random_validators() {
    let rng = &mut TestRng::default();

    // Construct the process.
    let process = Process::<CurrentNetwork>::load().unwrap();
    // Initialize a new finalize store.
    let (store, _temp_dir) = sample_finalize_store!();

    // Initialize the validators and delegators.
    let (validators, delegators) = initialize_stakers(&store, 1000, 1000,
↪   rng).unwrap();
    let mut validators = validators.into_iter();


    test_atomic_finalize!(store, FinalizeMode::RealRun, {




        for _ in 0..1000{

            let (validator_private_key_1, (validator_address_1, _, _,
↪   withdrawal_address_1)) = validators.next().unwrap();
            let (delegator_private_key, (delegator_address, _)) =
↪   delegators.first().unwrap();


            let delegator_amount = MIN_DELEGATOR_STAKE;

        bond_public(
            &process,
            &store,
            delegator_private_key,
            &validator_address_1,
            delegator_address,
            delegator_amount,
            rng,
        )
        .unwrap();
        unbond_public(&process, &store, &delegator_private_key,
↪   delegator_address, delegator_amount, 2, rng).unwrap();

        claim_unbond_public(&process, &store, &delegator_private_key,
↪   delegator_address, 363, rng).unwrap();
}

        Ok(())
    })
    .unwrap();
}
```

This test shows that a delegator is able to bond and unbond to 1000 (example value) random addresses permissionlessly.

To run it, place the above in `test_credits.aleo`.

cvetanovv

The main problem is that the `delegated[]` mapping is not cleaned up and will get very large over time, leading to higher computational costs.

But we would consider it a future integration issue. According to the Readme:

> Should potential issues, like broken assumptions about function behavior, be reported if they could pose risks in future integrations, even if they might not be an issue in the context of the scope? If yes, can you elaborate on properties/invariants that should hold?
>
> - No

There is also an example of this being done by a malicious user who intentionally filled the array. But here comes a lot of controversial stuff.

- What does the malicious user gain from this? He will have to do many transactions to eventually fill the array. Each transaction will cost him money without taking anything in return.
- Not everyone can do it.
- If it was so easy to fill a map or an array, then almost every contest would have a similar valid issue because there is almost no contest that doesn't iterate through an array.
- This is why the rules specify that it must be `time-sensitive`. Here are the DoS rules.
- If this happens naturally over time then according to the readme it is invalid because the protocol does not concerned about future integration issues.

sammy-tm

How is this a future integration issue? By this logic every issue is a future integration issue. Can you point out what exactly is getting "integrated" here? The exploit is possible right off the bat and doesn't require any future update to be available.

This is not the same as other Solidity contests where storage is filled up through mappings, etc. This is a layer 1 blockchain and the mapping here affects every single transaction in the network.

What benefit does the attacker have in doing this? An entire network is getting affected by this and I don't think the attacker needs to have a "benefit" from carrying out this attack. If attacks were only carried out if the attacker got a benefit, this would invalidate #35 and #34, because the cost is high and the benefit is 0 for both these issues.

Please look at this issue from PoV of making the network DoS resilient.

sammy-tm

And no, the mapping will not be filled automatically over time. The actor in question needs to purposely craft transactions with malicious intent. Why? Because honest delegators will only delegate to addresses that are either in the committee or will join the committee.

WangSecurity

I agree that it's not a future integration, but still unsure on the DOS part and the cost of the attack.

> Hence the impact is permanent.

So if this array is iterated in (almost) each transaction and it leads to a DOS, then if this attack is executed it would DOS the entire chain and the DOS would permanent. Hence, the only way to mitigate such attack if it were to happen is make a new network (yep, I'm exaggerating here, but want to understand how severe the impact here). Or it can be mitigated in any way if the attack were to happen?

And the cost would be just paying the fees for making the transaction to DOS the chain, correct?

sammy-tm

You're right, there is no way to mitigate this issue as it is as there is no way to clear the delegated[] mapping from storage. Hence impact is permanent.

Yes the cost is just gas fees. The credits can be reutilised.

sammy-tm

Not to be nit-picky but it is evident from the sponsors comment here that they don't want the storage to hold more data than it should.

https://github.com/sherlock-audit/2024-05-aleo-judging/issues/23#issuecomment-2199085149

However in that particular issue the `unbonding[]` mapping doesn't cause a storage issue as it requires a bond to remain active and is removed from storage once it's zero.

WangSecurity

Based on the above comments, I agree this issue is valid with medium severity. But, I'm planning to reject the escalation since the report doesn't clearly describe the permanent DOS and is a bit vague cause it says "potentially leading" and "may lead" to DOS, so I believe it wouldn't be unfair to penalise the Lead judge for classifying this report as low severity. Hope for the understanding on that decision.

sammy-tm

This is unfair because it's not possible to show the exact DoS because of test infrastructure limitation. The "vague" argument can be applied to #34 and #35 as well.

This is an edge case because in most contests the attack scenarios can be reproduced through tests, however it is not possible in this case as this is an audit of a layer 1 blockchain.

I request HoJ to value this edge case and reconsider their decision because this is definitely a valid issue and can cause permanent issues for the protocol. Invalidating this wouldn't be fair.

WangSecurity

I don't say the report is vague because it lacks POC, excuse me for the confusion. The problem is that with the report it's hard to verify the real world impact and how severe the DOS would be.

I believe I've said I'm planning to validate the issue with medium severity, but to reject the escalation. Excuse me if the comment was unclear and confusing. Are there additional duplicates of this issue or it's a unique?

sammy-tm

Yes, there are no duplicates.

As long as the issue turns valid, I am okay with the decision of rejecting the escalation.

infosecual

Good find Sammy

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- sammy-tm: accepted

Issue M-2: `unbond_public` logic causes issues for some delegators preventing partial withdrawals

Source: https://github.com/sherlock-audit/2024-05-aleo-judging/issues/31

Found by

dtheo

Summary

Delegators that share the same withdrawal address as a validator will be unable to make partial withdrawal due to a logic error in `unbond_public`. In some cases this can accidentally trigger the complete unbonding of the valdiator even when the delegator does not intend this.

Vulnerability Detail

If a validator ever attempts to delegate to itself or for any reason a delegator shares a withdrawal address with the validator it is bonded to then the logic in `unbond_public` will prevent it from making partial withdrawals.

Imagine the case:

1. A validator AAA is validating with 100 credits and a withdrawal address of BBB
2. A delegator CCC is bonded to validator AAA with 10M credits and also uses withdrawal address BBB
3. Delegator CCC attempts to unbond ANY amount of credits from the validator using `unbond_public`, lets say 1 microcredit.eg. delegator calls `unbond_public(stakers_addr = CCC, microcredits = 1)`

The following logic to determine if the delegators bond should be entirely removed will always incorrectly report that it should:

```
// Check if the delegator will fall below 10,000 bonded credits.
lt r16 10_000_000_000u64 into r17;

// If the validator is forcing the delegator to unbond OR the delegator will
↪  fall below 10,000 bonded credits.
or r11 r17 into r18;

// Determine the amount to unbond: requested amount if >= 10,000 credits,
↪  otherwise the full bonded amount.
ternary r18 r5.microcredits r2 into r19;
```

This is due to the fact that the logic to determine if the delegator is being force unbonded by the validator will always incorrectly report that it is. This will cause both the deleagator and the validator to completely unbond and force their credits into the unbonding period (360 blocks). This will happen even if the delegator calls `unbond_public` with 0 microcredits.

Impact

This can incorrectly force unbonding for delegators and validators.

Code Snippet

snarkVM/synthesizer/program/src/resources/credits.aleo#L487-L494)

I have also created an annotated pseudo code version of `unbond_public` to make it easier to read.

Tool used

Manual Review

Recommendation

Add a a conditional to check `r7` (this is set to true if caller == stakers withdrawal address). If this is set and a smaller amount of bond reduction is supplied than the total (as the function argument `microcredits` then consider this a delegator withdrawal and handle the arithmetic correctly. This will allow validators to still force unbond and will prevent delegators from accidentally unbonding everything without meaning to.

Discussion

evanmarshall

This is a real issue. I would categorize it as minor. Why it's minor is due to the factor that a single user (with both a validator and a delegator must set it up this way) and this single user has the ability to migrate to use separate withdrawal addresses.

sherlock-admin3

> Escalate
>
> The delegator setting the same withdraw address as the validator is essentially a self-rekt and therefore when this same withdraw address unbonds the delegator it is logically expected that the "validator unbond delegator" logic will apply here.

You've deleted an escalation for this issue.

evanmarshall

I'm not clear on the meaning of escalation here. Can you explain?

I think both interpretations are fair but I tend to side with the OP that you should be able to partial unbond. It is also a self rekt with very little consequence ie no funds lost or stolen. Only 360 blocks + one more transaction solve the issue.

infosecual

> Only 360 blocks + one more transaction solve the issue.

This downplays the possible impact a bit much in my opinion. This could force-remove a validator and its delegator without the operator meaning to. You can clean up the mess by changing your withdrawal address but there is no functionality to "just change withdrawal address". Depending how the delegator and validator are set up they might have to withdrawal the validator and all of its delegators, wait the unbonding period, then re-enroll them with different withdrawal addresses. They could not re-bond the new validator until all of the previous delegators that helped it get to the 10M credit minimum have done this.

infosecual

> I'm not clear on the meaning of escalation here. Can you explain?

Edit: Haxatron is escalating issues that he believes have been judged incorrectly. If he is right and can prove it he stands to be rewarded. https://docs.sherlock.xyz/audits/judging/escalation-period

Haxatron

> Haxatron is attempting to escalate to get valid issues marked as invalid. If he is successful and he can get some of his invalids changed to valid he stands to make more of the pot.

Firstly, this comment is uncalled for and I apologise if you feel this way. But I believe I have been quite fair in my escalations thus far. If you believe I have made any incorrect points you are free to dispute it.

Secondly, I still do not think this is a real issue because the delegator must set the same withdrawal address as the validator, for which there is no legitimate reason to do so. And then, when this validator withdrawal address which is the same as the delegator withdrawal address unbonds the delegator it is expected that the "validator unbonds delegator" logic applies.

morbsel

I agree with Haxatron, if the withdrawal is set to the same withdrawal address as the validator it means only the validator withdrawal address can unbond the delegator's bond and when the validator unbonds a delegator's bond the whole amount should get unbonded, stated at: https://github.com/sherlock-audit/2024-05-aleo/blob/55b2e4a02f27602a54c11f964f6f610fee6f4ab8/snarkVM/synthesizer/program/src/resources/credits.aleo#L415-L417 So partial unbond shouldn't even be a point of discussion.

infosecual

> Firstly, this comment is uncalled for and I apologise if you feel this way.

I apologize. I did not mean this in a condescending way or anything. I see how that came across now and I did not mean for it to sound this way. Please excuse me. This is better wording and is more in line with what I mean to say:

Haxatron is escalating issues that he believes have been judged incorrectly. If he is right and can prove it he stands to be rewarded.

evanmarshall

I still believe this issue as a whole remains an issue. In theory, a delegator should be able to partially unbond. It will be common for validator operators to handle large credit amounts by delegating to themselves. Validator keys are necessarily hot (why only the 100 credit self bond is required). Delegator keys and withdrawals addresses can be cold and use offline signing. It's possible that a validator tries to use the same withdrawal address for both validator and delegator. Through normal operation, a validator operator will want to withdraw rewards to fund operations. Partial unbonds would be expected in this case.

Haxatron

Since the sponsor has stated that there is a legitimate use case for a validator and a delegator to have the same withdrawal address, I do agree that this is a valid issue and will delete the escalation.

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.