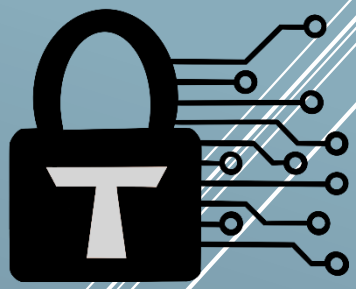
Trust Security

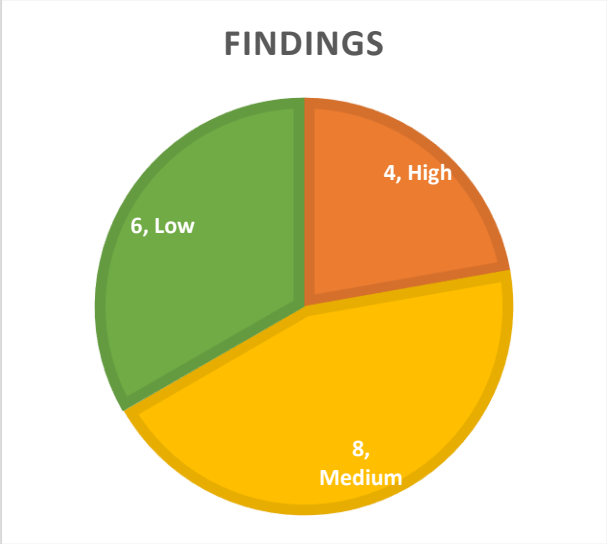


Smart Contract Audit

MorpheusAI Builders Migration

21/03/2025

Executive summary

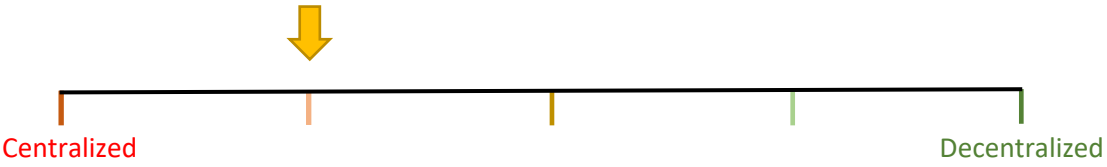


Category	Staking
Audited file count	2
Lines of Code	647
Auditor	Trust
Time period	26/02/25 - 04/03/25

Findings

Severity	Total	Open	Fixed	Acknowledged
High	4	-	4	-
Medium	8	-	5	3
Low	6	-	4	2

Centralization score



Signature

Digitally signed by Trust
Date: 2025.03.21 18:46:46 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	05/03/25	Client report
0.2	12/03/25	Mitigation review
0.3	20/03/25	Mitigation review
0.4	21/03/25	Mitigation review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- BuildersV2.sol
- BuilderSubnets.sol

Repository details

- **Repository URL:** https://github.com/MorpheusAIs/SmartContracts
- **Commit hash:** 57db1dbdaec4eb2f16e03cec1b8dd9a1394d85a9
- **Mitigation review commit hash:** c5ef0d0e5d24736c5d3ec1e791727e260766e0b3
- **Mitigation #2 review commit hash:** c62e556bb71fb9208afad014be32790b01cec602
- **Mitigation #3 review commit hash:** 98a5f3901fa07d41ec9d143f741643743902e9ae

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

About the Auditors

Trust has established a dominating presence in the smart contract security ecosystem since 2022. He is a resident on the Immunefi, Sherlock and C4 leaderboards and is now focused in auditing and managing audit teams under Trust Security. When taking time off auditing & bug hunting, he enjoys assessing bounty contests in C4 as a Supreme Court judge.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	Moderate	While most of the code is of reasonable complexity, the new rewards formula is very involved.
Documentation	Excellent	Project is well documented, although there are a few deviations from the code-level behavior.
Best practices	Good	Project mostly adheres to industry standards.
Centralization risks	Moderate	The migration process is trusted and centralized. All contracts are upgradeable by admin.

Findings

High severity findings

TRST-H-1 Deposits can be locked and yield rates would be incorrectly calculated due to double-counting of user's withdrawal

- **Category:** Accounting flaws
- **Source:** BuildersV2.sol
- **Status:** Fixed

Description

When the admin claims, the virtual bonus is instantly removed: The **totalVirtualDeposits** is reduced by the difference between virtual and true deposited amount. Then the builder pool **virtualDeposited** is set to the true deposited amount.

```
// Update pool data
totalPoolData.distributedRewards += newPoolRewards_;
totalPoolData.rate = currentRate_;
totalPoolData.totalVirtualDeposited =
    totalPoolData.totalVirtualDeposited +
    builderPoolData.deposited -
    builderPoolData.virtualDeposited;
// Update builder data
builderPoolData.rate = currentRate_;
builderPoolData.virtualDeposited = builderPoolData.deposited;
builderPoolData.pendingRewards = 0;
```

The issue is that there is no synchronization between these data structures and the per-user data structure (UserData). When a user withdraws, the **totalVirtualDeposited** and **virtualDeposited** are updated with the delta from old and new **virtualDeposited_**. This means if a withdrawal leaves 0 deposit, it will cause a decrement of the entire old **virtualDeposited** from the two data structures. But that doesn't make sense, because in *claim()* they were already reduced to account for only true deposited amount, so the result is a double-decrement of the bonus amount.

```
// Update pool data
totalPoolData.distributedRewards += newPoolRewards_;
totalPoolData.rate = currentRate_;
totalPoolData.totalDeposited = totalPoolData.totalDeposited + newDeposited_
- userData.deposited;
totalPoolData.totalVirtualDeposited =
    totalPoolData.totalVirtualDeposited +
    virtualDeposited_ -
    userData.virtualDeposited;
// Update builder data
builderPoolData.rate = currentRate_;
builderPoolData.pendingRewards = pendingRewards_;
builderPoolData.deposited = builderPoolData.deposited + newDeposited_ -
userData.deposited;
builderPoolData.virtualDeposited =
    builderPoolData.virtualDeposited +
    virtualDeposited_ -
    userData.virtualDeposited;
```

That means it is simple to show scenarios where the bad accounting causes withdrawals to revert:

1. Builders contract is deployed.
2. Builder pool is created.
3. 10 users deposit 1 MOR each, on 10x multiplier. **totalVirtualDeposited** = 100, **builderPoolData.deposited** = 10, **builderPoolData.virtualDeposited** = 100, 10 users have: **deposited** = 1, **virtualDeposited** = 10.
4. After **claimLockEnd**, admin calls *claim()*. **totalVirtualDeposited** = 100 + 10 - 100 = 10. **builderPoolData.virtualDeposited** = 10.
5. User 1 withdraws their token. **totalVirtualDeposited** = 10 + 0 - 10 = 0. **builderPoolData.virtualDeposited** = 10 + 0 - 10 = 0.
6. User 2 withdraws their token. **totalVirtualDeposited** = 0 + 0 - 10 = -10 -> revert.

The root cause is that the per-user **virtualDeposited** does not reflect the true amount a user is contributing to the builder and total **virtualDeposited**. It needs to be ignored and the user's deposited taken instead in this case, without applying a bonus.

Recommended mitigation

The current corrupted state can be fixed by an upgrade setting storage to fixed-up values. In terms of the root cause, either *claim()* should no longer reduce the **virtualDeposited** values, or additional accounting needs to be introduced so that not the entire user's **virtualDeposited** is decremented on withdrawals.

Team response

Fixed.

Mitigation review

The issue was addressed by identifying when a builder pool has already claimed, and reducing by the user's true deposited amount instead of the virtual deposited.

TRST-H-2 An attacker could make anyone lose their pending rewards

- **Category:** Logical flaws
- **Source:** BuilderSubnets.sol
- **Status:** Fixed

Description

In BuilderSubnets, rewards are calculated using the *getStakerRewards()* function.

```
function getStakerRewards(bytes32 subnetId_, address stakerAddress_, uint128
to_) public view returns (uint256) {
    Staker storage staker = stakers[subnetId_][stakerAddress_];
    uint256 from_ = (staker.lastInteraction == 0 ? block.timestamp :
staker.lastInteraction).max(
        rewardCalculationStartsAt
    );
    if (from_ >= block.timestamp) {
        return 0;
```

```
    }
    uint256 currentRewards_ = getPeriodRewardForStake(staker.virtualStaked,
uint128(from_), to_);
    return staker.pendingRewards + currentRewards_;
}
```

When the **from** parameter is the current block.timestamp or in the future, it does an early exit and early zero. But that ignores the already accrued rewards stored in **staker.pendingRewards**. By returning zero, user's potential earnings would be nullified. The zero **pendingRewards** would be stored under the user's account in *_updateStorage()*. An attacker could easily exploit it to force victim to lose all their rewards. For example, they could call *collectPendingRewards()* to accrue rewards into pendingRewards and set the last interaction to block.timestamp. Then, they could call *collectPendingRewards()* again to trigger the bug and set rewards to zero.

Recommended mitigation

In the affected line above, the function should return **staker.pendingRewards**.

Team response

The code was changed, this part is no longer necessary under the new model for calculating rewards.

Mitigation review

The affected line has been removed in the refactor.

TRST-H-3 Approved funds of Treasury to BuilderSubnets could be drained

- **Category:** Validation flaws
- **Source:** BuilderSubnets.sol
- **Status:** Fixed

Description

The *collectPendingRewards()* function was introduced to process rewards in smaller time windows, in case the gas cost of processing the entire time period is too high for a block.

```
function collectPendingRewards(
    bytes32 subnetId_,
    address stakerAddress_,
    uint128 to_
) external onlyExistedSubnet(subnetId_) {
    Staker storage staker = stakers[subnetId_][stakerAddress_];
    to_ = uint128(to_.min(block.timestamp));
    _updateStorage(subnetId_, stakerAddress_, staker.staked,
staker.claimLockEnd, to_);
    emit PendingRewardsCollected(subnetId_, stakerAddress_, staker);
}
```

Notice that **to** is validated to be below block.timestamp, but there is no lower bound set. It is passed into *_updateStorage()* which sets the last interaction timestamp to it.

```
// Update Staker data
staker.lastInteraction = interactionTimestamp_;
```

In fact, an attacker can abuse it to claim rewards multiple times over the same time segment. In *getStakerRewards()*, it trusts the **staker.lastInteraction** to be the last claimed timestamp, and accrues between that time and the requested time. Therefore, one could call *claim()* and *collectPendingRewards(to_ = veryLongAgo)* repeatedly to drain rewards from the treasury. All approved funds from the treasury to the BuilderSubnets contract would be stolen.

Recommended mitigation

Validate the **to** parameter to be higher than the existing interaction timestamp. That protects the invariant that any moment in time can only be credited once.

Team response

The code was changed, this part is no longer necessary under the new model for calculating rewards.

Mitigation review

The affected function has been entirely removed in the refactor.

TRST-H-4 Rewards will accrue from an unwanted time period at treasury's expense

- **Category:** Logical flaws
- **Source:** BuilderSubnets.sol
- **Status:** Fixed

Description

As part of the reward calculation refactor, *getStakerRewards()* changed and no longer uses the **rewardCalculationStartsAt** parameter. The value was not introduced in any other location in code, meaning it is ignored. Therefore, if rewards logic starts accruing from before this value, it will collect incorrect rewards from the treasury. In essence the treasury will pay twice for the same time period.

Recommended mitigation

In case the **from** timestamp is lower than **rewardCalculationStartsAt**, use that value instead. Also, make sure this value is set to avoid accepting rewards before the admin configures it.

Team response

Fixed.

Mitigation review

Issue has been addressed by enforcing rewards only start at **rewardCalculationStartsAt**.

Medium severity findings

TRST-M-1 The migration plan is not resistant to griefing attacks

- **Category:** DoS attacks
- **Source:** BuildersV2.sol
- **Status:** Acknowledged

Description

The migration plan is operated by the Builders owner. For each user, the admin will call *migrateUsersStake()* to stake the holdings on the Subnets contract. However, that plan is not resistant to griefing attacks. An adversary could create a very large amount of users or subnets, effectively forcing the admin to either not respect their migration plan and abandon certain users, or spent an exorbitant amount of gas to move all users over. Either option is a bad outcome.

Recommended mitigation

Refactor the migration process to support a self-served migration option. Then the admin could at their discretion move some or all of the users, but they are not forced to.

Team response

Added functionality where the user himself can transfer his deposit. For admin added possibility of batch transfer. Subnets will be created only not with zero deposits, if there will be a lot of subnets with low balance (attack), we will deal with it locally, maybe they will not be transferred.

Mitigation review

Users can now call *migrateUserStake()* on their own. While it fixes the current issue, it introduces another griefing risk. A user could make a migration of a large user batch fail by frontrunning the *migrateUsersStake()* call with *migrateUserStake()*. If they are part of the migration batch, it will attempt to migrate the same user again in the batch call, which will revert. Consider not reverting if a user has already migrated in *migrateUserStake()*, and simply returning from the function early.

Team Response

We'd rather have a more obvious answer in the form of an exception than just a return from a function. If the attacker wants to play this game, we'll play it, the migration will be done anyway.

TRST-M-2 Creation of subnets is vulnerable to frontrunning, disrupting the migration

- **Category:** Frontrunning attacks
- **Source:** BuilderSubnets.sol
- **Status:** Acknowledged

Description

As part of the migration, the owner will create subnets in BuilderSubnets which are equivalent to the ones in BuildersV2 and migrate users to them. However, once Subnets is deployed, anyone can register subnets with the original names from Builders, providing malicious builder parameters. Griefers can even frontrun the creation of new subnets when owner selects an alternative name for them. This would delay or block the migration process.

Recommended mitigation

Before the migration is over, block the creation of subnets by unprivileged users. After the migration, consider using the msg.sender as a salt as a way to generate the builder ID, and allow multiple builders with the same name.

Team response

Added a check when creating a subnet that migration should be completed. For griefing after migration, it will be addressed if needed.

Mitigation review

The check has been introduced successfully.

TRST-M-3 An attacker could make it impractical for a target victim to stake

- **Category:** Griefing attacks
- **Source:** BuilderSubnets.sol
- **Status:** Fixed

Description

The *collectPendingRewards()* function never validates that the passed staker address exists. If it doesn't, this allows an attacker to grief an address which is currently not a staker, but will be in the future (e.g. when frontrunning a *stake()* call). By calling the function with a timestamp far in the past, like 0, it would force the reward calculation to be done day-by-day, for the 55 years since the epoch. Even after fixing H-3 (historic timestamps), attackers could still call the function as soon as the contract is created to make unnecessary calculations whenever a staker comes in.

Recommended mitigation

Validate that the staker already holds a non-zero deposit in the contract.

Team response

The code was changed, this part is no longer necessary under the new model for calculating rewards.

Mitigation review

The function was removed during the refactor.

TRST-M-4 The rewarding formula leads to inefficient reward distribution

- **Category:** Leak of value issues

- **Source:** BuilderSubnets.sol
- **Status:** Acknowledged

Description

The reward formula operates by taking the rewards for time windows up to 1 day, from the last user interaction, dividing by the total rewards from the start to current time, and multiplying by the user's virtual stake. The method intends to cap the total rewards distributed to the linear distribution in buildersPoolData. However, that is a theoretic upper bound and cannot practically be reached:

- The virtual staked capacity is rising every block in a parabolic shape. If users don't immediately stake to match the capacity, the delta is going to waste, i.e. unclaimable.
- When users claim, their virtual stake from time **T** is the numerator, while the total rewards up to time **T + up to 1 day** is in the denominator. That means the claim cannot actually achieve full efficiency, and the best strategy is to claim at every block to reduce waste, but even that is not completely loss-free.

When considering the rewards for the entire linear distribution are set aside at the treasury, and previously they were all handed out, it is clear the current solution is more wasteful.

Recommended mitigation

Consider refactoring the reward mechanism to reduce complexity and improve efficiency. The current way it is done in BuildersV2 is in our opinion superior to the newer model.

Team response

This part of the audit is likely to be changed after the award mechanics are changed.

Mitigation review

The change reduces complexity by only accruing rewards on the total stake. However, the efficiency concerns outlined above remain as those concern the day-by-day calculation which still takes place.

Team Response

Acknowledged.

TRST-M-5 Migration could be tripped by users who fill the Subnets capacity

- **Category:** Logical flaws
- **Source:** BuilderSubnets.sol
- **Status:** Fixed

Description

During staking, the function ensures that the current staked amount does not exceed some capacity calculated for the moment in time. However, that logic can be used to disrupt the migration process. A user could make large deposits to Subnets after it is set up. Then, the loop migrating users from BuildersV2 would fail the *stake()* call.

Recommended mitigation

Disallow staking in Subnets except from the Builders contract, until the **isMigrationOver** flag is set to true.

Team response

Fixed.

Mitigation review

The suggested fix has been implemented.

TRST-M-6 Integrators of the Builders contract may lose critical functionality

- **Category:** Integration issues
- **Source:** BuildersV2.sol
- **Status:** Acknowledged

Description

As part of the migration process, the admin will create subnets in accordance with the builder groups in Builders. Users' stakes will then be transported to the matching subnet. The issue is that if the users or admins of subnets are smart contract integrations, this would lead to them being unable to interact with their stake or subnet. In the worst case, deposits would be unretrievable.

Recommended mitigation

The original API in Builders should still be supported. The contract will have special permissions to call functions in Subnets on behalf of their own caller.

Team response

I don't think we will run into this problem, at the very least the funds by other methods can be returned to the smart contract holders.

TRST-M-7 An attacker could significantly delay another user's claim time

- **Category:** Access-control issues
- **Source:** BuilderSubnets.sol
- **Status:** Fixed

Description

Before the migration, users are allowed to call *stake()* on behalf of another user in the Subnets contract. Supposedly, that is not harmful as they would be donating tokens to them. However, it was not considered that *stake()* also sets the **claimLockEnd** value. This means a user can stake as little as 1 wei after the migrator called *stake()* on some user, and pass a maximal claim lock time (it is capped to the subnet-level value). In this way, an attacker could significantly delay the time users wait until they can claim rewards.

Recommended mitigation

Only the BuildersV2 contract should be able to call *stake()* on behalf of another user.

Team response

Fixed.

Mitigation review

The issue has been addressed as suggested.

TRST-M-8 Functionality may be halted as reward calculation continuously reverts

- **Category:** Logical flaws
- **Source:** BuilderSubnets.sol
- **Status:** Fixed

Description

As part of the reward calculation refactor, incremental reward calculation was removed, so the entire rewards from the **from** to **to** timestamp are calculated in one go. If, for any reason, reward accrual did not occur for a long time, the calculation loop may cost more gas than what is available in a block. As it will only grow larger, all functionality will be frozen.

Recommended mitigation

Restore the backup functionality to add rewards incrementally.

Team response

Fixed.

Mitigation review

The team reinserted the *collectPendingRewards()* function that has been removed as part of the refactor. However, H-3 has not been addressed, so attacker could drain the rewards using a **to** parameter far in the past.

Also, in case **virtualStaked == 0**, the rewards for the period in time are forever unclaimed. Consider logging the amount of rewards not spent, so the treasury knows how much allocation is freed.

Team response

Fixed.

Mitigation review

Both issues have been addressed as recommended.

Low severity findings

TRST-L-1 The getter for staker's power factor could return wrong values

- **Category:** Logical flaws
- **Source:** BuilderSubnets.sol
- **Status:** Fixed

Description

Anyone can query the power factor of a staker using *getStakerPowerFactor()*. It calculates the lock multiplier from **lastStake** to **claimLockEnd**.

```
function getStakerPowerFactor(bytes32 subnetId_, address stakerAddress_)
public view returns (uint256) {
    if (!_subnetExists(subnetId_)) {
        return PRECISION;
    }
    Staker storage staker = stakers[subnetId_][stakerAddress_];
    return getPowerFactor(staker.lastStake, staker.claimLockEnd);
}
```

The implementation actually assumes there were no withdrawals since the **lastStake** timestamp, because if there were, the power factor would be recomputed at that point in time. Then the value returned from the function would be incorrect. That can be observed from the behavior of *_updateStorage()*:

```
if (newStaked_ != staker.staked) {
    BuildersSubnetData storage buildersSubnetData =
buildersSubnetsData[subnetId_];

    uint256 multiplier_ = getPowerFactor(interactionTimestamp_,
claimLockEnd_);
    uint256 newVirtualStaked_ = (newStaked_ * multiplier_) / PRECISION;
```

An integration using the function would be wrong some of the time at resolving the multiplier.

Recommended mitigation

Save the multiplier in storage, or use another method to calculate the multiplier.

Team response

Fixed.

Mitigation review

The multiplier is now calculated correctly using **virtualStaked** and **staked** values.

TRST-L-2 Divide-before-multiply in *getPeriodRewardForStake()* leads to precision loss

- **Category:** Precision loss errors
- **Source:** BuilderSubnets.sol
- **Status:** Fixed

Description

In Solidity calculations, it is generally recommended to multiply all operands before division. This pattern avoids amplifying the precision loss of the division rounding downwards. However, in the lines below in *getPeriodRewardForStake()* the practice isn't followed:

```
uint256 shareForPeriod_ = (virtualStaked_ * PRECISION) /
emissionToPeriodEnd_;
rewards_ += (shareForPeriod_ * emissionFromPeriodStartToPeriodEnd_) /
PRECISION;
```

Rewards would therefore be lower than they can be, although the amount is not significant.

Recommended mitigation

Multiply before dividing. In fact, there is no need to multiply by PRECISION and then divide by it.

Team response

Fixed.

Mitigation review

Issue has been addressed as suggested.

TRST-L-3 An unused contract could have approval to handle the Builders funds

- **Category:** Dangling approval issues
- **Source:** BuildersV2.sol
- **Status:** Fixed

Description

The *setBuilderSubnets()* function is used to load a new subnets contract to the legacy Builders contract. Note that it performs a max approval to the new builderSubnets. Nothing prevents the function from being used multiple times, for example to move from a faulty subnets contract to a correct one. However, the previous contract would still have the maximum approval, which introduces risks of abuse.

Recommended mitigation

Reset the approval to the previous **builderSubnets** before setting it to a new value.

Team response

Fixed.

Mitigation review

Issue has been addressed as suggested.

TRST-L-4 The reward distribution can change retroactively against expectations

- **Category:** Time-sensitivity issues

- **Source:** BuildersV2.sol
- **Status:** Acknowledged

Description

The owner is free to change the builder reward distribution through *setBuildersPoolData()*. The issue is that changes apply retroactively, meaning users that expect a certain reward to have already built up for them, could get less rewards.

Recommended mitigation

Ensure that only a single buildersPoolData can ever be set, or that the next one is mutually exclusive with the first in its time period.

Team response

The main contract that distributes awards in the mainnet could potentially change the award pool data. This was done by analogy.

TRST-L-5 Fees are rounded in favor of the user against standard practices

- **Category:** Rounding issues
- **Source:** BuilderSubnets.sol
- **Status:** Fixed

Description

It is standard practice to round in favor of the protocol, in order to reduce risks of value loss or precision-loss exploits. However, that is not followed in the functions below:

```
function _getProtocolFee(uint256 amount_, bytes32 operation_) private view
returns (uint256, address) {
    (uint256 feePart_, address treasuryAddress_) =
IFeeConfig(feeConfig).getFeeAndTreasuryForOperation(
        address(this),
        operation_
    );
    uint256 fee_ = (amount_ * feePart_) / PRECISION;
    return (fee_, treasuryAddress_);
}
function _getSubnetFee(uint256 amount_, bytes32 subnetId_) private view
returns (uint256, address) {
    BuildersSubnet storage subnet = buildersSubnets[subnetId_];
    uint256 fee_ = (amount_ * subnet.fee) / PRECISION;
    return (fee_, subnet.feeTreasury);
}
```

The lines highlighted round down, but the fee should round up.

Recommended mitigation

Round up the protocol and subnet fees.

Team response

Fixed.

Mitigation review

Issue has been addressed as suggested.

TRST-L-6 Power factor reset is manual and calculates per-staker, which has risks

- **Category:** DoS attacks
- **Source:** BuilderSubnets.sol
- **Status:** Acknowledged

Description

After discussion with the team, it is desired that power factor would no longer be applied for staking after claim end time. The *resetPowerFactor()* will reset it for any lock that completed. However, there are two limitations with the manual handling:

- An attacker could make it expensive for the admin to reset their power factor by splitting their holdings to arbitrarily small amounts, each behind a separate account.
- Between the time the lock ends and the time *resetPowerFactor()* is executed, the lock owner has gained more rewards than intended.

Recommended mitigation

Either acknowledge the limitations, or refactor the code so that the user's multiplier is fixed without admin intervention.

Team response

Acknowledged.

Additional recommendations

TRST-R-1 Enforce valid state

It is possible to improve the validation of state.

- The **rewardCalculationStartsAt** variable is only set by a custom setter, but when consuming it, it is never checked to be non-zero.
- The *setIsMigrationOver()* function should only be able to set migration from false to true.
- Unpausing in the BuildersV2 contract should be disabled after migration. The user-functions should never be used.

TRST-R-2 Early return can improve performance

In *getPeriodRewardForStake()*, the code performs relatively intense calculations. In case the **virtualStaked_** is zero, it is guaranteed to result in no rewards, so it should skip the expensive loop.

TRST-R-3 Improve docs accuracy

Documentation says: *"The multiplier changes each time the user stakes, withdraws tokens or claim."* However, during *claim()*, the multiplier logic is skipped in *updateStorage()*.

Centralization risks

TRST-CR-1 Contract upgradeability risk

Both the Subnets and BuildersV2 contracts are upgradeable by the owner. Naturally, that means that a compromised admin account could take over all funds and approvals of the contract. Any other owner-related risk in the contract is overshadowed by the upgrade risk.

TRST-CR-2 Migration process is trusted

The migration is completely done by the owner. That means users must trust the owner to faithfully transfer their funds to the new contract.

Systemic risks

TRST-SR-1 Rewards are assumed to be available in the Treasury

In order to satisfy claims, the Subnets contract uses approved funds from the Treasury. If the linear distribution rewards are not available, claiming will fail.