Forta Delegated Staking

Date	November 2022
Auditors	George Kobakhidze, Tejaswa Rastogi

1 Executive Summary

This report presents the results of our engagement with the **Forta Foundation** to review the **Forta Delegated Staking system of contracts**.

The review was conducted over 33 calendar days from **Monday, October 31st**, to **Friday, December 2nd**, by **George Kobakhidze** and **Tejaswa Rastogi**.

The project repository is well organized, modular, and has plentiful comments that allow the reader to better understand the intent of the contracts, not just their input and output parameters.

Off-chain logic and operational management are important pieces of this system that are out of scope for this audit. This assessment was focused solely on the smart contracts that are listed in the Scope.

The audit started by performing a general analysis of the system and its logic. The assessment concerned the individual contracts to be deployed, their interactions, implementation (proxy or not) details, configuration updates for things like access and roles, identification of trusted roles and addresses, outlining the layers of logical separation, and so on. This led the audit team to better understand how the different pieces of the system stand on their own and connect to each other.

During the second half of the audit the above learnings and created supporting documentation, like the system diagram, were used to hone in on the the specific Forta delegated staking and rewards distribution use cases with the Scanner, Agent, and NodeRunner registries. Function control and data flows were identified and analyzed from their beginning in the FortaStaking contract to the individual updates to Accumulator's rewards accumulation storage for various subject IDs.

Ultimately, this audit identified issues some of which occur due to high modularity of such systems that utilize many similar components with small, although crucial, differences. In spite of impact of some of the issues, they do not indicate flawed core design but do showcase the emergence of possible edge cases and operational difficulty to keep up with the system's complexity as more variants of its components are introduced.

2 Scope

Our review focused on the commit hash `50a985c4ca106840f38b2dcb728e00c50ebc7b26` of the Forta Contracts repository. The list of files in scope can be found in the Appendix.

2.1 Objectives

Together with the **Forta Foundation** team, we identified the following priorities for our review:

1. Assess if the system is implemented consistently with the intended functionality, and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.
3. Review the Staking, Delegation, and Rewarding logic.
4. Assess if there are any unexpected downtime edge cases for bots and scanners.
5. The Scanner, Agent, and Node Runner registries are transferable where appropriate.
6. Architecture overview of the Forta system.

3 System Overview

The Forta Delegated Staking system is a significantly sized complex net of contracts and operational tasks, and, although some contracts could be looked at in isolation, the majority of the logic needs to be analyzed from the point of view of the whole system.

It is important to note that this repository can be used to build generic staking for the Forta platform, and, in fact, could be adjusted for use for other platforms as well. While there is logic specific to Forta (like the agent registries and the software update signaller) and the FORTA token, these contracts could allow for any ERC-20 token staking and delegation of staking shares. Similarly, for example, the registries of staking subjects are separate contracts and have layers of separation from the main staking contract itself, like the `SubjectGateway` contract, allowing for modular updates, changes, or even full replacement of these registries, depending on the use case (proxy upgrades notwithstanding).

Below is a short description for each of the main contracts and a simple overall architecture diagram.

3.1 AccessManager

This contract is responsible for verifying if the users calling contracts have appropriate roles for what they want to execute, which is done through the `hasRole` function and `onlyRole` modifier. Every contract that inherits `BaseComponentUpgradeable` calls the

`AccessManager` contract for access control.

3.2 FortaStaking

This is the main staking contract where users and other contracts call into. It begins the actual staking process and holds the staking token as well. Primarily, it calls into the `StakeSubjectGateway` contract to interact with the registries and the `StakeAllocator` contract to allocate shares for the rewards.

3.3 StakeAllocator & RewardsDistributor

These two contracts contain the logic for storing and calculating the allocation of rewards based on staking shares that the users receive. The `RewardsDistributor` contract is also important for the claim and distribution of staking rewards. Similarly to other contracts, the `StakeAllocator` contract interacts with the `StakeSubjectGateway` to call into the appropriate registries.

3.4 SlashingController

This is a contract that users and administrators of the system may use to slash malicious actors and arbitrate disputes. Since the slashing process needs a failsafe to halt malicious actors, this contract may freeze their stake, and so it also calls into the main `FortaStaking` contract. Like others, it also interacts with the `StakeSubjectGateway` contract.

3.5 StakeSubjectGateway

One of the more important contracts in the Forta system, the `StakeSubjectGateway` contract is called into by the majority of the system. Based on the arguments provided to its functions, it is able to determine what kind of subject registry needs to be called into to receive information relevant to the system at that time. It acts as a layer between most contracts and the registries. Thus, it calls into those subject registries directly through the `IDelegatedStakeSubject` and `IDirectStakeSubject` interfaces coupled with the registries addresses. This contract also calls into the `FortaStaking` contract to determine the staking balances of users.

3.6 NodeRunnerRegistry, AgentRegistry, ScannerRegistry

These contracts are the registries that maintain the logic for the tokens crucial to the Forta system as they represent the staking subject and their associated nodes, such as Scanners, Agents, and NodeRunner nodes after the Scanner migration. They hold the logic to determine whether a subject is registered, operational, enabled, what kind of metadata it has, how to update that metadata and so on. They don't perform many external calls with the exceptions of the `NodeRunnerRegistry` calling the `StakeAllocator` to determine allocated stake for managed subjects, and the `ScannerRegistry` calling the `NodeRunnerRegistry` for after the migration process.

3.7 ScannerToNodeRunnerMigration

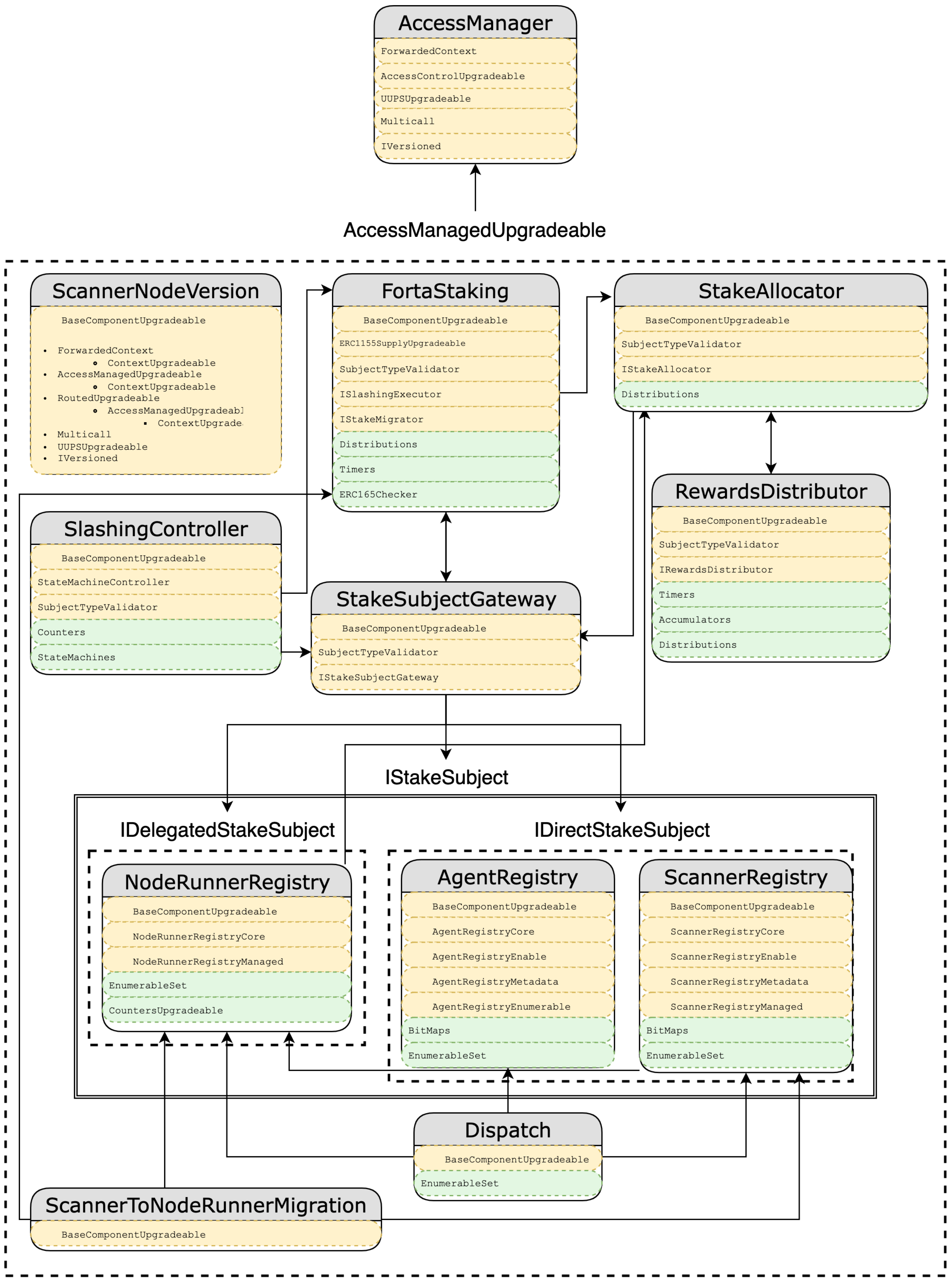
This is a contract to facilitate the migration from the `ScannerRegistry` to the `NodeRunnerRegistry`. It calls into both of the mentioned registries as well as the main staking contract `FortaStaking`.

3.8 Dispatch

This contract is responsible for providing Forta system jobs and linking the Agents to the Scanners that are being run. As such, it calls into all of the relevant subject registries - `ScannerRegistry`, `NodeRunnerRegistry`, and `AgentRegistry`.

3.9 ScannerNodeVersion

This contract stands slightly isolated and is used by the Forta Foundation team (or the privileged address with the appropriate role) to signal that a new update is out for the Forta software. It does not call into any contracts, outside of its base calling into the `AccessManager` like all other contracts.



4 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation.

4.1 Actors

The Forta Staking system has several actors that interact with each other.

The Forta Council

The team that develops, deploys, and has the `OWNER` role for the Forta staking system of contracts ("the Forta Council") itself is an important actor for the health of the system. As an owner of the smart contracts, this team also holds the power to assign and take on at least the following roles:

- `DEFAULT_ADMIN_ROLE` - Grants/revokes roles and performs administrative operations.
- `ENS_MANAGER_ROLE` - Sets up the ENS reverse registration name.
- `UPGRADER_ROLE` - Authorizes contract upgrades.
- `AGENT_ADMIN_ROLE` - Enables/disables agents, sets stake threshold, sets frontrunning delay.
- `SCANNER_ADMIN_ROLE` - Enables/disables agents, sets stake threshold for a `chainID`.
- `NODE_RUNNER_ADMIN_ROLE` - Sets managed stake threshold for a `chainID`, sets registration delay.
- `SCANNER_2_NODE_RUNNER_MIGRATOR_ROLE` - Handles migrations from scanners to node runners, deregisters scanner nodes, registers migrated scanners/node runner nodes.
- `DISPATCHER_ROLE` - Links/unlinks an agent to a scanner.
- `MIGRATION_EXECUTOR_ROLE` - Migrates from the old `ScannerRegistry` NFTs to a single `NodeRunnerRegistry` NFT.
- `SLASHER_ROLE` - Executes a slashing proposal, authorizes reverts of slashing proposals if they have already been reviewed, slashes/freezes/unfreezes withdrawals of a subject stake.
- `SWEEPER_ROLE` - Sweeps all the unwanted tokens.

- `REWARDER_ROLE` - Manages rewards for the node runner subjects.
- `SLASHING_ARBITER_ROLE` - Dismisses/rejects/reviews a slashing proposal.
- `STAKING_CONTRACT_ROLE` - Allocates stake on deposit for a `DELEGATED` subject.
- `STAKING_ADMIN_ROLE` - Sets deposit amount for slashing proposals, sets slash percent and penalities, sets delegation params.
- `ALLOCATOR_CONTRACT_ROLE` - Allocates/unallocates/transfers shares.
- `SCANNER_VERSION_ROLE` - Signals scanner nodes to update their binaries with latest version.
- `SCANNER_BETA_VERSION_ROLE` - Signals scanner nodes to update their binaries with latest beta version.

The Node/Scanner/Agent owners

Perhaps most important actors in the system are the owners of the nodes and tokens that perform and are responsible for the security scanning activity as per the Forta configurations. They not only directly bring value to the system by using it, but also collect rewards from staking and delegated staking.

Forta stakers

With introduction of delegated staking, users may delegate their stake to actors that actually interact with the system, thus giving regular users an ability to earn yield on their tokens in return for trusting the subjects that they stake on.

4.2 Trust Model

The Forta contracts define a bunch of roles that are deemed to manage various sets of operations, like slashing, sweeping, rewards, migration, upgrades, version updates, and more. These roles, as described above, are assigned by the smart contract owners, i.e. the Forta Council. This makes the system more centralized and also error-prone, as there will always be a risk of private key losses or accidentally providing a role to an unintentional address.

Similarly, the responsibilities associated with these roles may be complex to manage. Functions for slashing arbitration and rewards, for example, need to be executed with correct parameters or the system may run into issues. We recommend reducing the reliance on centralized roles and the number of powerful roles, thus making the system more decentralized and robust.

Finally, due to the upgradeable proxy functionality of the system, there is an additional risks for incorrect deployments. For example, as noted many times in the code comments and documentation, as the upgrades to the smart contracts add functionality, they also add complexity to the storage layout, which may introduce storage collisions. This is currently helped by the `__gap` variable within the smart contracts to have a buffer for storage slots.

It is important to note that the Forta ecosystem, including the Forta Council, Forta Foundation, and other actors, are in the ongoing process of decentralization. This includes both standard security practices such as multi-sigs, public token-voting with solutions like Snapshot, robust governance processes, and other forms of decentralizing the decision making processes. The previous paragraphs intend to describe the security configuration and trust model associated with having admin and power roles in the contracts. This is not a commentary on the general decentralization of power and tokens in the Forta ecosystem.

The delegated stake subjects are a source of a significant trust assumptions for stakers as well. While this trust is rewarded with yield, staking users should be aware that this yield does not come for free, and their stake may in fact be slashed if they delegate to someone who performs a slashable action.

4.3 Additional Notes

During the audit, the Forta Foundation team independently identified issues to be remediated and beneficial changes to be made to the Forta system. These are:

Renaming `NodeRunnerRegistry` to `ScannerPoolRegistry`.

`NodeRunnerRegistry` should be called `ScannerPoolRegistry` to avoid confusion with the intent of this registry. The associated PR can be found here: PR#139

Registering or enabling a scanner node can take nodes in the `NodeRunnerRegistry` offline.

Registering or enabling a scanner node can potentially make all of the nodes in a `NodeRunnerRegistry` appear as not operational. The intended path is to stake as much as you need then register the scanners. However, a user could mess up, and that could take out a big player. Therefore, the Forta Foundation team would feel safer if users can't do those actions unless they have enough staking, so a check should be added. The associated PR can be found here: PR#140

`NodeRunnerRegistry.getManagedStakeThreshold()` incorrect implementation.

The `NodeRunnerRegistry` has a function to retrieve the managed stake threshold for a specific managed subject ID through `getManagedStakeThreshold()` function. It does so by calling a mapping on the managed ID - `_scannerStakeThresholds[managedId]`. However, the actual mapping for the stake threshold is not set on managed subject IDs. Instead, it is set on the chain ID associated with that subject ID. To retrieve that chain ID, another mapping needs to be retrieved - `_nodeRunnerChainId[managedId]`. So, the correct implementation of this would be `_scannerStakeThresholds[_nodeRunnerChainId[managedId]]`. The associated PR can be found here: PR#145.

Adjusting the visibility of address variables in the `FortaStaking` contract.

For certain off chain components to work, they need to retrieve addresses of other contracts through the main `FortaStaking` contract. Specifically, the `StakeAllocator` address needs to be retrievable, so there is a change to make its visibility public. Additionally, the `RewardsDistributor` contract address is not used in the `FortaStaking` contract, so the address variable for that is removed altogether. The associated commits can be found here: Commit#1, Commit#2.

5 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.

- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major 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.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

5.1 didTransferShares function has no access control modifier Critical ✓ Fixed

Resolution

The concerned function has now been restricted to be only called by `STAKING_CONTRACT_ROLE` in a pull request 146 with final commit hash as `97fbd425b64d793252f39d94b378e2655286d947`

Description

The staked tokens (shares) in Forta are meant to be transferable. Similarly, the rewards allocation for these shares for delegated staking is meant to be transferable as well. This allocation for the shares' owner is tracked in the `StakeAllocator`. To enable this, the Forta staking contract `FortaStaking` implements a `_beforeTokenTransfer()` function that calls `_allocator.didTransferShares()` when it is appropriate to transfer the underlying allocation.

code/contracts/components/staking/FortaStaking.sol:L572-L585

```
function _beforeTokenTransfer(
    address operator,
    address from,
    address to,
    uint256[] memory ids,
    uint256[] memory amounts,
    bytes memory data
) internal virtual override {
    for (uint256 i = 0; i < ids.length; i++) {
        if (FortaStakingUtils.isActive(ids[i])) {
            uint8 subjectType = FortaStakingUtils.subjectTypeOfShares(ids[i]);
            if (subjectType == DELEGATOR_NODE_RUNNER_SUBJECT && to != address(0) && from != address(0)) {
                _allocator.didTransferShares(ids[i], subjectType, from, to, amounts[i]);
            }
```

Due to this, the `StakeAllocator.didTransferShares()` has an `external` visibility so it can be called from the `FortaStaking` contract to perform transfers. However, there is no access control modifier to allow *only* the staking contract to call this. Therefore, anyone can call this function with whatever parameters they want.

code/contracts/components/staking/allocation/StakeAllocator.sol:L341-L349

```
function didTransferShares(
    uint256 sharesId,
    uint8 subjectType,
    address from,
    address to,
    uint256 sharesAmount
) external {
    _rewardsDistributor.didTransferShares(sharesId, subjectType, from, to, sharesAmount);
}
```

Since the allocation isn't represented as a token standard and is tracked directly in the `StakeAllocator` and `RewardsDistributor`, it lacks many standard checks that would prevent abuse of the function. For example, this function does not have a check for allowance or `msg.sender==from`, so any user could call `didTransferShares()` with `to` being their address and `from` being any address they want to transfer allocation from, and the call would succeed.

Recommendation

Apply access control modifiers as appropriate for this contract, for example `onlyRole()`.

5.2 Incorrect reward epoch start date calculation Major ✓ Fixed

Resolution

The suggested recommendations have been implemented in a pull request 144 with a final hash as `b23ffa370596e614c813bd3b882f3d8c6d15067e`

Description

The Forta rewards system is based on epochs. A privileged address with the role `REWARDER_ROLE` calls the `reward()` function with a parameter for a specific `epochNumber` that consequently distributes the rewards for that epoch. Additionally, as users stake and delegate their stake, accounts in the Forta system accrue weight that is based on the active stake to distribute these rewards. Since accounts can modify their stake as well as delegate or un-delegate it, the rewards weight for each account can be modified, as seen, for example, in the `didAllocate()` function. In turn, this modifies the `DelegatedAccRewards` storage struct that stores the accumulated rewards for each share id. To keep track of changes done to the accumulated rewards, epochs with checkpoints are used to manage the accumulated rate of rewards, their value at the checkpoint, and the timestamp of the checkpoint.

For example, in the `didAllocate()` function the `addRate()` function is being called to modify the accumulated rewards.

code/contracts/components/staking/rewards/RewardsDistributor.sol:L89-L101

```
function didAllocate(
    uint8 subjectType,
    uint256 subject,
    uint256 stakeAmount,
    uint256 sharesAmount,
    address staker
) external onlyRole(ALLOCATOR_CONTRACT_ROLE) {
    bool delegated = getSubjectTypeAgency(subjectType) == SubjectStakeAgency.DELEGATED;
    if (delegated) {
        uint8 delegatorType = getDelegatorSubjectType(subjectType);
        uint256 shareId = FortaStakingUtils.subjectToActive(delegatorType, subject);
        DelegatedAccRewards storage s = _rewardsAccumulators[shareId];
        s.delegated.addRate(stakeAmount);
```

Then the function flow goes into `setRate()` that checks the existing accumulated rewards storage and modifies it based on the current timestamp.

code/contracts/components/staking/rewards/Accumulators.sol:L34-L36

```
function addRate(Accumulator storage acc, uint256 rate) internal {
    setRate(acc, latest(acc).rate + rate);
}
```

code/contracts/components/staking/rewards/Accumulators.sol:L42-L50

```
function setRate(Accumulator storage acc, uint256 rate) internal {
    EpochCheckpoint memory ckpt = EpochCheckpoint({ timestamp: SafeCast.toUint32(block.timestamp), rate: SafeCast.toUint224(rate), val
    uint256 length = acc.checkpoints.length;
    if (length > 0 && isCurrentEpoch(acc.checkpoints[length - 1].timestamp)) {
        acc.checkpoints[length - 1] = ckpt;
    } else {
        acc.checkpoints.push(ckpt);
    }
}
```

Namely, it pushes epoch checkpoints to the list of account checkpoints based on its timestamp. If the last checkpoint's timestamp is during the current epoch, then the last checkpoint is replaced with the new one altogether. If the last checkpoint's timestamp is different from the current epoch, a new checkpoint is added to the list. However, the `isCurrentEpoch()` function calls a function `getCurrentEpochTimestamp()` that incorrectly determines the start date of the current epoch. In particular, it doesn't take the offset into account when calculating how many epochs have already passed.

code/contracts/components/staking/rewards/Accumulators.sol:L103-L110

```
function getCurrentEpochTimestamp() internal view returns (uint256) {
    return ((block.timestamp / EPOCH_LENGTH) * EPOCH_LENGTH) + TIMESTAMP_OFFSET;
}

function isCurrentEpoch(uint256 timestamp) internal view returns (bool) {
    uint256 currentEpochStart = getCurrentEpochTimestamp();
    return timestamp > currentEpochStart;
}
```

Instead of `((block.timestamp / EPOCH_LENGTH) * EPOCH_LENGTH) + TIMESTAMP_OFFSET`, it should be `(((block.timestamp - TIMESTAMP_OFFSET) / EPOCH_LENGTH) * EPOCH_LENGTH) + TIMESTAMP_OFFSET`. In fact, it should simply call the `getEpochNumber()` function that correctly provides the epoch number for any timestamp.

code/contracts/components/staking/rewards/Accumulators.sol:L95-L97

```
function getEpochNumber(uint256 timestamp) internal pure returns (uint32) {
    return SafeCast.toUint32((timestamp - TIMESTAMP_OFFSET) / EPOCH_LENGTH);
}
```

In other words, the resulting function would look something like the following:

```
    function getCurrentEpochTimestamp() public view returns (uint256) {
        return (getEpochNumber(block.timestamp) * EPOCH_LENGTH) + TIMESTAMP_OFFSET;
    }
```

Otherwise, if `block.timestamp` is such that `(block.timestamp - TIMESTAMP_OFFSET) / EPOCH_LENGTH = n` and `block.timestamp / EPOCH_LENGTH = n+1`, which would happen on roughly 4 out of 7 days of the week since `EPOCH_LENGTH = 1 weeks` and `TIMESTAMP_OFFSET = 4 days`, this would cause the `getCurrentEpochTimestamp()` function to return the *end timestamp* of the epoch (which is in the future) instead of the start. Therefore, if a checkpoint with such a timestamp is committed to the account's accumulated rewards checkpoints list, it will always fail the below check in the epoch it got submitted, and any checkpoint committed afterwards but during the same epoch with a similar type of `block.timestamp` (i.e. satisfying the condition at the beginning of this paragraph), would be pushed to the top of the list instead of replacing the previous checkpoint.

code/contracts/components/staking/rewards/Accumulators.sol:L45-L48

```
if (length > 0 && isCurrentEpoch(acc.checkpoints[length - 1].timestamp)) {
    acc.checkpoints[length - 1] = ckpt;
} else {
    acc.checkpoints.push(ckpt);
```

This causes several checkpoints to be stored for the same epoch, which would cause issues in functions such as `getAtEpoch()`, that feeds into `getValueAtEpoch()` function that provides data for the rewards' share calculation. In the end, this would cause issues in the accounting for the rewards calculation resulting in incorrect distributions.

During the discussion with the Forta Foundation team, it was additionally discovered that there are edge cases around the limits of epochs. Specifically, epoch's end time and the subsequent epoch's start time are exactly the same, although it should be that it is only the start of the next epoch. Similarly, that start time isn't recognized as part of the epoch due to `>` sign instead of `>=`. In particular, the following changes need to be made:

```
    function getEpochEndTimestamp(uint256 epochNumber) public pure returns (uint256) {
        return ((epochNumber + 1) * EPOCH_LENGTH) + TIMESTAMP_OFFSET - 1; <---- so it is 23:59:59 instead of next day 00:00:00
    }

    function isCurrentEpoch(uint256 timestamp) public view returns (bool) {
        uint256 currentEpochStart = getCurrentEpochTimestamp();
        return timestamp >= currentEpochStart; <--- for the first second on Monday
    }
```

Recommendation

A refactor of the epoch timestamp calculation functions is recommended to account for:

- The correct epoch number to calculate the start and end timestamps of epochs.
- The boundaries of epochs coinciding.
- Clarity in functions' intent. For example, adding a function just to calculate any epoch's start time and renaming `getCurrentEpochTimestamp()` to `getCurrentEpochStartTimestamp()`.

5.3 A single unfreeze dismisses all other slashing proposal freezes Major ✓ Fixed

Resolution

As per the recommendation, the Forta team modified the logic in favor of open proposals. Now, every `shareId` will have a counter for open proposals, which will be incremented every time a new proposal is launched and will be unfrozen only if the counter is zero. The changes were implemented in a pull request 149 with a final hash `76338b1417bdb7b1da49b7e74ad011b307907f7f`

Description

In order to retaliate against malicious actors, the Forta staking system allows users to submit slashing proposals that are guarded by submitting along a deposit with a slashing reason. These proposals immediately freeze the proposal's subject's stake, blocking them from withdrawing that stake.

At the same time, there can be multiple proposals submitted against the same subject, which works out with freezing – the subject remains frozen with each proposal submitted. However, once any one of the active proposals against the subject gets to the end of its lifecycle, be it `REJECTED`, `DISMISSED`, `EXECUTED`, or `REVERTED`, the subject gets unfrozen altogether. The other proposals might still be active, but the stake is no longer frozen, allowing the subject to withdraw it if they would like.

In terms of impact, this allows bad actors to avoid punishment intended by the slashes and freezes. A malicious actor could, for example, submit a faulty proposal against themselves in the hopes that it will get quickly rejected or dismissed while the existing, legitimate proposals against them are still being considered. This would allow them to get unfrozen quickly and withdraw their stake. Similarly, in the event a bad staker has several proposals against them, they could withdraw right after a single slashing proposal goes through.

Examples

code/contracts/components/staking/slashing/SlashingController.sol:L174-L179

```
function dismissSlashProposal(uint256 _proposalId, string[] calldata _evidence) external onlyRole(SLASHING_ARBITER_ROLE) {
    _transition(_proposalId, DISMISSED);
    _submitEvidence(_proposalId, DISMISSED, _evidence);
    _returnDeposit(_proposalId);
    _unfreeze(_proposalId);
}
```

code/contracts/components/staking/slashing/SlashingController.sol:L187-L192

```
function rejectSlashProposal(uint256 _proposalId, string[] calldata _evidence) external onlyRole(SLASHING_ARBITER_ROLE) {
    _transition(_proposalId, REJECTED);
    _submitEvidence(_proposalId, REJECTED, _evidence);
    _slashDeposit(_proposalId);
    _unfreeze(_proposalId);
}
```

code/contracts/components/staking/slashing/SlashingController.sol:L215-L229

```
function reviewSlashProposalParameters(
    uint256 _proposalId,
    uint8 _subjectType,
    uint256 _subjectId,
    bytes32 _penaltyId,
    string[] calldata _evidence
) external onlyRole(SLASHING_ARBITER_ROLE) onlyInState(_proposalId, IN_REVIEW) onlyValidSlashPenaltyId(_penaltyId) onlyValidSubjectTyp
    // No need to check for proposal existence, onlyInState will revert if _proposalId is in undefined state
    if (!subjectGateway.isRegistered(_subjectType, _subjectId)) revert NonRegisteredSubject(_subjectType, _subjectId);

    _submitEvidence(_proposalId, IN_REVIEW, _evidence);
    if (_subjectType != proposals[_proposalId].subjectType || _subjectId != proposals[_proposalId].subjectId) {
        _unfreeze(_proposalId);
        _freeze(_subjectType, _subjectId);
    }
```

code/contracts/components/staking/slashing/SlashingController.sol:L254-L259

```
function revertSlashProposal(uint256 _proposalId, string[] calldata _evidence) external {
    _authorizeRevertSlashProposal(_proposalId);
    _transition(_proposalId, REVERTED);
    _submitEvidence(_proposalId, REVERTED, _evidence);
    _unfreeze(_proposalId);
}
```

code/contracts/components/staking/slashing/SlashingController.sol:L267-L272

```
function executeSlashProposal(uint256 _proposalId) external onlyRole(SLASHER_ROLE) {
    _transition(_proposalId, EXECUTED);
    Proposal memory proposal = proposals[_proposalId];
    slashingExecutor.slash(proposal.subjectType, proposal.subjectId, getSlashedStakeValue(_proposalId), proposal.proposer, slashPercen
    slashingExecutor.freeze(proposal.subjectType, proposal.subjectId, false);
}
```

code/contracts/components/staking/slashing/SlashingController.sol:L337-L339

```
function _unfreeze(uint256 _proposalId) private {
    slashingExecutor.freeze(proposals[_proposalId].subjectType, proposals[_proposalId].subjectId, false);
}
```

Recommendation

Introduce a check in the unfreezing mechanics to first ensure there are no other active proposals for that subject.

5.4 Storage gap variables slightly off from the intended size Medium ✓ Fixed

Resolution

The Forta Team worked on the storage layout to maintain a consistent storage buffer in the inheritance tree. The changes were made through multiple pull requests, also an easy-to-understand layout description has been added through a pull request 157. However, we still found some inconsistencies and recommend doing a thorough review of the buffer space again.

For instance, in FortaStaking (considering the latest commit)

```
uint64 private _withdrawalDelay;

// treasury for slashing
address private _treasury;
```

the above-mentioned storage variables will be taking a single slot, however, separate slots are considered for the buffer space(referring to the storage layout description to determine `__gap` buffer).

Description

The Forta staking system is using upgradeable proxies for its deployment strategy. To avoid storage collisions between contract versions during upgrades, `uint256[] private __gap` array variables are introduced that create a storage buffer. Together with contract state variables, the storage slots should sum up to 50. For example, the `__gap` variable is present in the `BaseComponentUpgradeable` component, which is the base of most Forta contracts, and there is a helpful comment in `AgentRegistryCore` that describes how its relevant `__gap` variable size was calculated:

code/contracts/components/BaseComponentUpgradeable.sol:L62

```
uint256[50] private __gap;
```

code/contracts/components/agents/AgentRegistryCore.sol:L196

```
uint256[41] private __gap; // 50 - 1 (frontRunningDelay) - 3 (_stakeThreshold) - 5 StakeSubjectUpgradeable
```

However, there are a few places where the `__gap` size was not computed correctly to get the storage slots up to 50. Some of these are:

code/contracts/components/scanners/ScannerRegistry.sol:L234

```
uint256[49] private __gap;
```

code/contracts/components/dispatch/Dispatch.sol:L333

```
uint256[47] private __gap;
```

code/contracts/components/node_runners/NodeRunnerRegistryCore.sol:L452

```
uint256[44] private __gap;
```

While these still provide large storage buffers, it is best if the `__gap` variables are calculated to hold the same buffer within contracts of similar types as per the initial intentions to avoid confusion.

During conversations with the Forta Foundation team, it appears that some contracts like `ScannerRegistry` and `AgentRegistry` should instead add up to 45 with their `__gap` variable due to the `StakeSubject` contracts they inherit from adding 5 from themselves. This is something to note and be careful with as well for future upgrades.

Recommendation

Provide appropriate sizes for the `__gap` variables to have a consistent storage layout approach that would help avoid storage issues with future versions of the system.

5.5 AgentRegistryCore - Agent Creation DoS Medium ✓ Fixed

Resolution

The Forta team as per the recommendations modified the minting logic to allow users to mint an `agentId` only for their own address in a pull request 155 with final hash as `7426891222e2bcdf2bbbec669905d5041f9fb58e`. Also, the team claims that the Agent Ids are generated through the Forta Bot SDK to minimize the collision risk. However, this has not been verified by the auditing team.

We still recommend notifying users to check whether an ID is already registered prior to making any commitment if a front-running delay is enabled, to avoid unintended DoS.

Description

AgentRegistryCore allows anyone to mint an `agentID` for the desired owner address. However, in some cases, it may fall prey to DoS, either deliberately or unintentionally.

For instance, let's assume the Front Running Protection is disabled or the `frontRunningDelay` is 0. It means anyone can directly create an agent without any prior commitment. Thus, anyone can observe pending transactions and try to front run them to mint an `agentID` prior to the victim's restricting it to mint a desired `agentID`.

Also, it may be possible that a malicious actor succeeds in frontrunning a transaction with manipulated data/chainIDs but with the same owner address and `agentID`. There is a good chance that victim still accepts the attacker's transaction as valid, even though its own transaction reverted, due to the fact that the victim is still seeing itself as the owner of that ID.

Taking an instance where let's assume the frontrunning protection is enabled. Still, there is a good chance that two users vouch for the same `agentIDs` and commits in the same block, thus getting the same frontrunning delay. Then, it will be a game of luck, whoever creates that agent first will get the ID minted to its address, and the other user's transaction will be reverted wasting the time they have spent on the delay.

As the `agentIDs` can be picked by users, the chances of collisions with an already minted ID will increase over time causing unnecessary reverts for others.

Adding to the fact that there is no restriction for owner address, anyone can spam mint any `agentID` to any address for any profitable reason.

Examples

code/contracts/components/agents/AgentRegistryCore.sol:L68-L77

```
function createAgent(uint256 agentId, address owner, string calldata metadata, uint256[] calldata chainIds)
public
    onlySorted(chainIds)
    frontrunProtected(keccak256(abi.encodePacked(agentId, owner, metadata, chainIds)), frontRunningDelay)
{
    _mint(owner, agentId);
    _beforeAgentUpdate(agentId, metadata, chainIds);
    _agentUpdate(agentId, metadata, chainIds);
    _afterAgentUpdate(agentId, metadata, chainIds);
}
```

Recommendation

1. Modify function `prepareAgent` to not commit an already registered `agentID`.
2. A better approach could be to allow sequential minting of `agentIDs` using some counters.
3. Only allow users to mint an `agentID`, either for themselves or for someone they are approved to.

5.6 Lack of checks for rewarding an epoch that has already been rewarded Medium ✓ Fixed

Resolution

The suggested recommendations have been implemented in a pull request 150 with final hash `76e1ae8ca16c92851f2bafb905f0e0c86542027c`. The reward logic has been modified to register the reward for an epoch only once and revert if called twice.

Description

To give rewards to the participating stakers, the Forta system utilizes reward epochs for each `shareId`, i.e. a delegated staking share. Each epoch gets their own reward distribution, and then `StakeAllocator` and `RewardsDistributor` contracts along with the Forta staking shares determine how much the users get.

To actually allocate these rewards, a privileged account with the role `REWARDER_ROLE` calls the `RewardsDistributor.reward()` function with appropriate parameters to store the `amount` a `shareId` gets for that specific `epochNumber`, and then adds the `amount` to the `totalRewardsDistributed` contract variable for tracking. However, there is no check that the `shareId` already received rewards for that `epoch`. The new reward amount simply replaces the old reward amount, and `totalRewardsDistributed` gets the new `amount` added to it anyway. This causes inconsistencies with accounting in the `totalRewardsDistributed` variable.

Although `totalRewardsDistributed` is essentially isolated to the `sweep()` function to allow transferring out the reward tokens without taking away those tokens reserved for the reward distribution, this still creates an inconsistency, albeit a minor one in the context of the current system.

Similarly, the `sweep()` function deducts the `totalRewardsDistributed` amount instead of the amount of *pending* rewards only. In other words, either there should be a different variable that tracks only pending rewards, or the `totalRewardsDistributed` should have token amounts deducted from it when users execute the `claimRewards()` function. Otherwise, after a few epochs there will be a really large `totalRewardsDistributed` amount that might not reflect the real amount of pending reward tokens left on the contract, and the `sweep()` function for the reward token is likely to fail for any amount being transferred out.

Examples

code/contracts/components/staking/rewards/RewardsDistributor.sol:L155-L167

```
function reward(
    uint8 subjectType,
    uint256 subjectId,
    uint256 amount,
    uint256 epochNumber
) external onlyRole(REWARDER_ROLE) {
    if (subjectType != NODE_RUNNER_SUBJECT) revert InvalidSubjectType(subjectType);
    if (!_subjectGateway.isRegistered(subjectType, subjectId)) revert RewardingNonRegisteredSubject(subjectType, subjectId);
    uint256 shareId = FortaStakingUtils.subjectToActive(getDelegatorSubjectType(subjectType), subjectId);
    _rewardsPerEpoch[shareId][epochNumber] = amount;
    totalRewardsDistributed += amount;
    emit Rewarded(subjectType, subjectId, amount, epochNumber);
}
```

Recommendation

Implement checks as appropriate to the `reward()` function to ensure correct behavior of `totalRewardsDistributed` tracking. Also, implement necessary changes to the tracking of pending rewards, if necessary.

5.7 Reentrancy in FortaStaking during ERC1155 mints Medium ✓ Fixed

Resolution

The Forta team implemented a Reentrancy Guard in a pull request 151 with a final hash `62080c17e9bd2be8105bfe4e59f36fad7be60fe5`

Description

In the Forta staking system, the staking shares (both “active” and “inactive”) are represented as tokens implemented according to the `ERC1155` standard. The specific implementation that is being used utilizes a smart contract acceptance check `_doSafeTransferAcceptanceCheck()` upon mints to the recipient.

code/contracts/components/staking/FortaStaking.sol:L54

```
contract FortaStaking is BaseComponentUpgradeable, ERC1155SupplyUpgradeable, SubjectTypeValidator, ISlashingExecutor, IStakeMigrator {
```

The specific implementation for `ERC1155SupplyUpgradeable` contracts can be found here, and the smart contract check can be found here.

This opens up reentrancy into the system’s flow. In fact, the reentrancy occurs on all mints that happen in the below functions, and it happens before a call to another Forta contract for allocation is made via either `_allocator.depositAllocation` or `_allocator.withdrawAllocation`:

code/contracts/components/staking/FortaStaking.sol:L273-L295

```
function deposit(
    uint8 subjectType,
    uint256 subject,
    uint256 stakeValue
) external onlyValidSubjectType(subjectType) notAgencyType(subjectType, SubjectStakeAgency.MANAGED) returns (uint256) {
    if (address(subjectGateway) == address(0)) revert ZeroAddress("subjectGateway");
    if (!subjectGateway.isStakeActivatedFor(subjectType, subject)) revert StakeInactiveOrSubjectNotFound();
    address staker = _msgSender();
    uint256 activeSharesId = FortaStakingUtils.subjectToActive(subjectType, subject);
    bool reachedMax;
    (stakeValue, reachedMax) = _getInboundStake(subjectType, subject, stakeValue);
    if (reachedMax) {
        emit MaxStakeReached(subjectType, subject);
    }
    uint256 sharesValue = stakeToActiveShares(activeSharesId, stakeValue);
    SafeERC20.safeTransferFrom(stakedToken, staker, address(this), stakeValue);

    _activeStake.mint(activeSharesId, stakeValue);
    _mint(staker, activeSharesId, sharesValue, new bytes(0));
    emit StakeDeposited(subjectType, subject, staker, stakeValue);
    _allocator.depositAllocation(activeSharesId, subjectType, subject, staker, stakeValue, sharesValue);
    return sharesValue;
}
```

code/contracts/components/staking/FortaStaking.sol:L303-L326

```
function migrate(
    uint8 oldSubjectType,
    uint256 oldSubject,
    uint8 newSubjectType,
    uint256 newSubject,
    address staker
) external onlyRole(SCANNER_2_NODE_RUNNER_MIGRATOR_ROLE) {
    if (oldSubjectType != SCANNER_SUBJECT) revert InvalidSubjectType(oldSubjectType);
    if (newSubjectType != NODE_RUNNER_SUBJECT) revert InvalidSubjectType(newSubjectType);
    if (isFrozen(oldSubjectType, oldSubject)) revert FrozenSubject();

    uint256 oldSharesId = FortaStakingUtils.subjectToActive(oldSubjectType, oldSubject);
    uint256 oldShares = balanceOf(staker, oldSharesId);
    uint256 stake = activeSharesToStake(oldSharesId, oldShares);
    uint256 newSharesId = FortaStakingUtils.subjectToActive(newSubjectType, newSubject);
    uint256 newShares = stakeToActiveShares(newSharesId, stake);

    _activeStake.burn(oldSharesId, stake);
    _activeStake.mint(newSharesId, stake);
    _burn(staker, oldSharesId, oldShares);
    _mint(staker, newSharesId, newShares, new bytes(0));
    emit StakeDeposited(newSubjectType, newSubject, staker, stake);
    _allocator.depositAllocation(newSharesId, newSubjectType, newSubject, staker, stake, newShares);
}
```

code/contracts/components/staking/FortaStaking.sol:L365-L387

```
function initiateWithdrawal(
    uint8 subjectType,
    uint256 subject,
    uint256 sharesValue
) external onlyValidSubjectType(subjectType) returns (uint64) {
    address staker = _msgSender();
    uint256 activeSharesId = FortaStakingUtils.subjectToActive(subjectType, subject);
    if (balanceOf(staker, activeSharesId) == 0) revert NoActiveShares();
    uint64 deadline = SafeCast.toUint64(block.timestamp) + _withdrawalDelay;

    _lockingDelay[activeSharesId][staker].setDeadline(deadline);

    uint256 activeShares = Math.min(sharesValue, balanceOf(staker, activeSharesId));
    uint256 stakeValue = activeSharesToStake(activeSharesId, activeShares);
    uint256 inactiveShares = stakeToInactiveShares(FortaStakingUtils.activeToInactive(activeSharesId), stakeValue);
    SubjectStakeAgency agency = getSubjectTypeAgency(subjectType);
    _activeStake.burn(activeSharesId, stakeValue);
    _inactiveStake.mint(FortaStakingUtils.activeToInactive(activeSharesId), stakeValue);
    _burn(staker, activeSharesId, activeShares);
    _mint(staker, FortaStakingUtils.activeToInactive(activeSharesId), inactiveShares, new bytes(0));
    if (agency == SubjectStakeAgency.DELEGATED || agency == SubjectStakeAgency.DELEGATOR) {
        _allocator.withdrawAllocation(activeSharesId, subjectType, subject, staker, stakeValue, activeShares);
    }
```

Although this doesn't seem to be an issue in the current Forta system of contracts since the allocator's logic doesn't seem to be manipulable, this could still be dangerous as it opens up an external execution flow.

Recommendation

Consider introducing a reentrancy check or emphasize this behavior in the documentation, so that both other projects using this system later and future upgrades along with maintenance work on the Forta staking system itself are implemented safely.

5.8 Unnecessary code blocks that check the same condition Minor ✓ Fixed

Resolution

The code block has been refactored under a single conditional block as per the suggested recommendation in a pull request 152 with a final hash as `0031cbdeb9450b86c49dd2c284efbe7af0eac542`

Description

In the `RewardsDistributor` there is a function that allows to set delegation fees for a `NodeRunner`. It adjusts the `fees[]` array for that node as appropriate. However, during its checks, it performs the same check twice in a row.

Examples

code/contracts/components/staking/rewards/RewardsDistributor.sol:L259-L264

```
if (fees[1].sinceEpoch != 0) {
    if (Accumulators.getCurrentEpochNumber() < fees[1].sinceEpoch + delegationParamsEpochDelay) revert SetDelegationFeeNotReady();
}
if (fees[1].sinceEpoch != 0) {
    fees[0] = fees[1];
}
```

Recommendation

Consider refactoring this under a single code block.

5.9 Event spam in RewardsDistributor.claimRewards Minor ✓ Fixed

Resolution

Forta team has implemented the recommended check in a pull request 153, as:
`if (epochRewards == 0) revert ZeroAmount("epochRewards");`

The implemented check will now be reverting the transaction if there exists no reward for an epoch number. However, it may not be a gas-efficient approach for the user claiming rewards and accidentally passing an incorrect epoch number. A better approach could be to transfer any reward and emit any event only for a non-zero epochReward.

Description

The `RewardsDistributor` contract allows users to claim their rewards through the `claimRewards()` function. It does check to see whether or not the user has already claimed the rewards for a specific epoch that they are claiming for, but it does not check to see if the user has any associated rewards at all. This could lead to event `ClaimedRewards` being spammed by malicious users, especially on low gas chains.

Examples

code/contracts/components/staking/rewards/RewardsDistributor.sol:L224-L229

```
for (uint256 i = 0; i < epochNumbers.length; i++) {
    if (_claimedRewardsPerEpoch[shareId][epochNumbers[i]][_msgSender()]) revert AlreadyClaimed();
    _claimedRewardsPerEpoch[shareId][epochNumbers[i]][_msgSender()] = true;
    uint256 epochRewards = _availableReward(shareId, isDelegator, epochNumbers[i], _msgSender());
    SafeERC20.safeTransfer(rewardsToken, _msgSender(), epochRewards);
    emit ClaimedRewards(subjectType, subjectId, _msgSender(), epochNumbers[i], epochRewards);
```

Recommendation

Add a check for rewards amounts being greater than 0.

5.10 SubjectTypes.sol files unused Minor ✓ Fixed

Resolution

The unused file has now been removed in commit 2548e0a4f7b38926362a759f4fa0611394348d6e

Description

There is a rogue file `SubjectTypes.sol` that is not being utilized. It appears that its intended functionality is being done by the `SubjectTypeValidator.sol` file as it even has a contract with the same name implemented there.

Examples

code/contracts/components/staking/SubjectTypes.sol:L4-L10

```
pragma solidity ^0.8.9;

uint8 constant SCANNER_SUBJECT = 0;
uint8 constant AGENT_SUBJECT = 1;
uint8 constant NODE_RUNNER_SUBJECT = 3;

contract SubjectTypeValidator {
```

Recommendation

Remove the `SubjectTypes.sol` file.

5.11 Lack of a check for the subject’s stake for reviewSlashProposalParameters Minor ✓ Fixed

Resolution

The recommended check has now been added in a pull request 154 with final hash as `17a75b2d67cfa7b5662d9f7b4e3cab8cb9c7907e`

Description

In the `SlashingController` contract, the address with the `SLASHING_ARBITER_ROLE` may call the `reviewSlashProposalParameters()` function to adjust the slashing proposal to a new `_subjectId` and `_subjectType`. However, unlike in the `proposeSlash()` function, there is no check for that subject having any stake at all.

While it may be assumed that the review function will be called by a privileged and knowledgeable actor, this additional check may avoid accidental mistakes.

Examples

code/contracts/components/staking/slashing/SlashingController.sol:L153

```
if (subjectGateway.totalStakeFor(_subjectType, _subjectId) == 0) revert ZeroAmount("subject stake");
```

code/contracts/components/staking/slashing/SlashingController.sol:L226-L229

```
if (_subjectType != proposals[_proposalId].subjectType || _subjectId != proposals[_proposalId].subjectId) {
    _unfreeze(_proposalId);
    _freeze(_subjectType, _subjectId);
}
```

Recommendation

Add a check for the new subject having stake to slash.

5.12 Comment and code inconsistencies Minor ✓ Fixed

Resolution

The comments have now been found fixed as per the implemented logic, primarily in the pull request 156 and in another commit with hash f4ee799ee192084965643b09b69f3cbeababd5ae

Description

During the audit a few inconsistencies were found between what the comments say and what the implemented code actually did.

Examples

Subject Type Agency for Scanner Subjects

In the `SubjectTypeValidator`, the comment says that the `SCANNER_SUBJECT` is of type `DIRECT` agency type, i.e. it can be directly staked on by multiple different stakers. However, we found a difference in the implementation, where the concerned subject is defined as type `MANAGED` agency type, which says that it cannot be staked on directly; instead it's a delegated type and the allocation is supposed to be managed by its manager.

code/contracts/components/staking/SubjectTypeValidator.sol:L21

```
* - SCANNER_SUBJECT --> DIRECT
```

code/contracts/components/staking/SubjectTypeValidator.sol:L66-L67

```
} else if (subjectType == SCANNER_SUBJECT) {
    return SubjectStakeAgency.MANAGED;
```

Dispatch refers to ERC721 tokens as ERC1155

One of the comments describing the functionality to `link` and `unlink` agents and scanners refers to them as ERC1155 tokens, when in reality they are ERC721.

code/contracts/components/dispatch/Dispatch.sol:L179-L185

```
/**
 * @notice Assigns the job of running an agent to a scanner.
 * @dev currently only allowed for DISPATCHER_ROLE (Assigner software).
 * @dev emits Link(agentId, scannerId, true) event.
 * @param agentId ERC1155 token id of the agent.
 * @param scannerId ERC1155 token id of the scanner.
 */
```

NodeRunnerRegistryCore comment that implies the reverse of what happens

A comment describing a helper function that returns address for a given scanner ID describes the opposite behavior. It is the same comment for the function just above that actually does what the comment says.

code/contracts/components/node_runners/NodeRunnerRegistryCore.sol:L259-L262

```
/// Converts scanner address to uint256 for FortaStaking Token Id.
function scannerIdToAddress(uint256 scannerId) public pure returns (address) {
    return address(uint160(scannerId));
}
```

ScannerToNodeRunnerMigration comment that says that no NodeRunner tokens must be owned

For the migration from Scanners to NodeRunners, a comment in the beginning of the file implies that for the system to work correctly, there must be no NodeRunner tokens owned prior to migration. After a conversation with the Forta Foundation team, it appears that this was an early design choice that is no longer relevant.

code/contracts/components/scanners/ScannerToNodeRunnerMigration.sol:L69

```
* @param nodeRunnerId If set as 0, a new NodeRunnerRegistry ERC721 will be minted to nodeRunner (but it must not own any prior),
```

code/contracts/components/scanners/ScannerToNodeRunnerMigration.sol:L91

```
* @param nodeRunnerId If set as 0, a new NodeRunnerRegistry ERC721 will be minted to nodeRunner (but it must not own any prior),
```

Recommendation

Verify the operational logic and fix either the concerned comments or defined logic as per the need.

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
code/contracts/components/BaseComponentUpgradeable.sol	`13680a70a63eb9d5c934c20b734a3d88b5672834`
code/contracts/components/Roles.sol	`0653e7dc8e3cb38d3b4f6d71f96da22a14a88776`
code/contracts/components/access/AccessManager.sol	`25615dd10406d5d4b29acd6382d35c9d1181a80f`
code/contracts/components/agents/AgentRegistry.sol	`cf64e86bc416c8922c2080f25d6230ac001f1315`
code/contracts/components/agents/AgentRegistryCore.sol	`d2f2b6cf51be66f4ba6f51fac15078c93d7c2775`
code/contracts/components/agents/AgentRegistryEnable.sol	`4613c92ae05d7967c6e57d8c72169725641c5549`
code/contracts/components/agents/AgentRegistryEnumerable.sol	`ddaa1eb3af25b88cb8b37a6ff7dd8846bb6350c6`
code/contracts/components/agents/AgentRegistryMetadata.sol	`6bd9965f3c5a5535ca1cb03638787cb704ede475`
code/contracts/components/dispatch/Dispatch.sol	`61823c3266bb7dce491016c15dfc815db01dd7b8`
code/contracts/components/node_runners/NodeRunnerRegistry.sol	`048ea91cd2a92da58dce8c17af7ac300a34b887e`
code/contracts/components/node_runners/NodeRunnerRegistryCore.sol	`fce62dc14d1c4c583da7f0e7a23b104d7793cb05`
code/contracts/components/node_runners/NodeRunnerRegistryManaged.sol	`5213b3f671831e1fadb42f2746b84e263966a0a4`
code/contracts/components/scanners/ScannerNodeVersion.sol	`78c54880c1a79f56238e452df695aa91471b9464`
code/contracts/components/scanners/ScannerRegistry.sol	`0ee261ce7fd5e3d70cc4f7a41d08aa012d57d0cf`
code/contracts/components/scanners/ScannerRegistryCore.sol	`b2b698264aa931747a2f4cebc584283c357c6cb4`
code/contracts/components/scanners/ScannerRegistryEnable.sol	`c60bad089815a8c46c1339a41550f6962ed82abe`
code/contracts/components/scanners/ScannerRegistryManaged.sol	`f4d4a85e41225de2b2099691b064bc4155acf7b6`
code/contracts/components/scanners/ScannerRegistryMetadata.sol	`d30a793132d6efc5d6d07bf21746fc375c3ca093`
code/contracts/components/scanners/ScannerToNodeRunnerMigration.sol	`de4fe62381307782b141f20d865e1c8cbf95bfd7`
code/contracts/components/staking/FortaStaking.sol	`476ce510d8777040a72be5c31ae65bfca566520b`
code/contracts/components/staking/FortaStakingUtils.sol	`29291cf1d741b5e95ac743e6da345e881e66e0d8`
code/contracts/components/staking/IStakeMigrator.sol	`1a2660dc3e30738c3f3b9a0e25ef83367b64e3b2`
code/contracts/components/staking/SubjectTypeValidator.sol	`00e678d3760ae1f87dff0a0f9d7d98563c2cea2e`
code/contracts/components/staking/SubjectTypes.sol	`64298bd5a3020e68c9cafa48235a2cd31ef3b6c6`
code/contracts/components/staking/allocation/IStakeAllocator.sol	`f35a3552ba58626631c8a49520f9f33cbed5ba36`
code/contracts/components/staking/allocation/StakeAllocator.sol	`c5290cd0c4d0e1fef8b975ecf76174fc713178b0`
code/contracts/components/staking/rewards/Accumulators.sol	`e7374d3556511447cdf1b00c1a753b3583ed8bf0`
code/contracts/components/staking/rewards/IRewardReceiver.sol	`8f99f8997f1f5277ef9f2ee2f61ae32d0be86b75`
code/contracts/components/staking/rewards/IRewardsDistributor.sol	`06c8789a9b0b9aea7e89728088f690da11515054`
code/contracts/components/staking/rewards/RewardsDistributor.sol	`26e217ce89290e93241b471825a17926335d0718`
code/contracts/components/staking/slashing/ISlashingExecutor.sol	`3e5be697de5ae84e17337e0270708a4f4d122c84`
code/contracts/components/staking/slashing/SlashReasons.sol	`7ea601dc64a46db02d7d25b0f2e349a542004eb3`
code/contracts/components/staking/slashing/SlashingController.sol	`aded81ebc24be0fdb1a4f351b7dbf0941b6d005d`
code/contracts/components/staking/stake_subjects/DelegatedStakeSubject.sol	`928c4b72f7c4ea5f57aad163179b4b2e1b421a44`

File	SHA-1 hash
code/contracts/components/staking/stake_subjects/DirectStakeSubject.sol	`45eb4e187547fcdc46634640fdbe54b164937d51`
code/contracts/components/staking/stake_subjects/IDelegatedStakeSubject.sol	`a1d9e6e796b84097fd87ab090336403b5b111f95`
code/contracts/components/staking/stake_subjects/IDirectStakeSubject.sol	`49cdf505f1b3b3ae227817bdebfc159cadbbb6a9`
code/contracts/components/staking/stake_subjects/IStakeSubject.sol	`331078471f24112cecca1dd5f1162fd84e9a35af`
code/contracts/components/staking/stake_subjects/IStakeSubjectGateway.sol	`8a65138fb2bb43a6578e4c7e94346ee96f5bb7ce`
code/contracts/components/staking/stake_subjects/StakeSubjectGateway.sol	`3e2a8b594d527005618f65b654acd3b2625708e4`
code/contracts/components/utils/AccessManaged.sol	`c3e6e3d0994f575d1566bccc6812dc3c8d89b223`
code/contracts/components/utils/ForwardedContext.sol	`5af82c66070302c2ac5891d361ad19c79d7b4b04`
code/contracts/components/utils/IVersioned.sol	`512877729a5e5223d57e219164d715f9e6c5d3d5`
code/contracts/components/utils/Routed.sol	`c1b4dc914d6950414a5f75e0435d6c0ea76c6253`
code/contracts/components/utils/StateMachines.sol	`2df1e5c7e860032d9538431dbe87b413f95eb92c`
code/contracts/errors/GeneralErrors.sol	`62a44c6dd0d2941e991e79b136ca9d4d28e28674`
code/contracts/token/Forta.sol	`f3770bf069882291bef2e8bf89d1aa64bcc9a40a`
code/contracts/token/FortaBridgedPolygon.sol	`bf069925d545e442076850b4b0289fd81785de15`
code/contracts/token/FortaCommon.sol	`763e1bb902ff4a7ce2da39099191d2a0c0ec2ad8`
code/contracts/tools/Distributions.sol	`7e1ab43835a220da0629a67aa5730fd795614cb9`
code/contracts/tools/ENSReverseRegistration.sol	`95623ba7b059747fd31215500b2111185cf5d57c`
code/contracts/tools/FrontRunningProtection.sol	`f6e33c05da28ef77093305fc3f6b8481342915bc`

Appendix 2 - Disclosure

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.