Introduction

The Forta team asked us to review and audit their Forta Token smart contracts. We looked at the code and now publish our results.

Scope

We audited commit `92d7a7ddd6672a7530a4bfc532d0d697e7f12744` of the forta-protocol/forta-token repository. The scope included the files inside the `contracts` directory, excluding those inside the `contracts/mocks`, `contracts/components/agents`, `contracts/components/_deprecated`, and `contracts/components/old` directories.

In summary, the files in scope were:

- `./components/access/AccessManager.sol`
- `./components/BaseComponentUpgradeable.sol`
- `./components/dispatch/Dispatch.sol`
- `./components/metatx/Forwarder.sol`
- `./components/Roles.sol`
- `./components/router/IRouter.sol`
- `./components/router/Router.sol`
- `./components/scanners/ScannerNodeVersion.sol`
- `./components/scanners/ScannerRegistry.sol`
- `./components/scanners/ScannerRegistryCore.sol`
- `./components/scanners/ScannerRegistryEnable.sol`

- ./components/scanners/ScannerRegistryManaged.sol
- ./components/scanners/ScannerRegistryMetadata.sol
- ./components/staking/FortaStaking.sol
- ./components/staking/FortaStakingSubjectTypes.sol
- ./components/staking/FortaStakingUtils.sol
- ./components/staking/IStakeController.sol
- ./components/utils/AccessManaged.sol
- ./components/utils/ForwardedContext.sol
- ./components/utils/Routed.sol
- ./components/utils/StakeAware.sol
- ./env/index.sol
- ./helpers/BatchRelayer.sol
- ./token/Forta.sol
- ./token/FortaBridgedPolygon.sol
- ./token/FortaCommon.sol
- ./tools/Distributions.sol
- ./tools/ENSReverseRegistration.sol
- ./tools/FrontRunningProtection.sol
- ./tools/FullMath.sol
- ./vesting/escrow/StakingEscrow.sol
- ./vesting/escrow/StakingEscrowFactory.sol
- ./vesting/escrow/StakingEscrowUtils.sol
- ./vesting/IRootChainManager.sol
- ./vesting/VestingWallet.sol
- ./vesting/VestingWalletV2.sol

All other project's files and directories (including tests), along with external dependencies and projects, game theory, and incentive design, were also excluded from the scope of this audit. External code and contract dependencies were assumed to work as documented.

System overview

Forta is meant to be a permissionless network of threat detection based on the usage of agents, scanners, and analyzers over a multi-chain platform. Agents will be developed by either independent developers or by security researchers, to then be used by scanners that will report alerts once they detect a potential threat to a project.

In order to improve the cost efficiency and scalability, the protocol deploys in L2 a set of contracts that implements functionalities to allow the staking of FORT tokens to earn rewards and the bridging of un-vested tokens in L1. However, if a subject (either a scanner or an agent) misbehaves or acts in a malicious way, the protocol could slash their stake in the platform and, with it, the stake that holders have on top of such subject.

To represent the stake in the protocol, the ERC1155 was used allowing to easily mint new shares for a subject and convert them into inactive shares once the holder initiates the withdrawal procedure.

Privileged roles

In the protocol there are several roles that have access to perform sensitive actions. In particular:

The `DEFAULT_ADMIN_ROLE` role can:

- Set new roles in the `AccessManager` contract.
- Set the agent and scanner registries in the `Dispatch` contract.
- Set the withdrawal delay in the `FortaStaking` contract.
- Set the treasury address in the `FortaStaking` contract.
- Set the minimum stake for subject type in the `FortaStaking` contract.
- Set the URI in the `FortaStaking` contract.
- Set the `AccessManager` contract's address in all `AccessManagedUpgradeable` contract's child.
- Set the `Router` contract's address in all `Routed` contract's child.
- Set the `StakeController` contract's address in all `StakeAware` contract's child.

The `ROUTER_ADMIN_ROLE` role can:

- Set the routing table for the `hookHandler` function.

The `ENS_MANAGER_ROLE` role can:

- Set the ENS reverse registration name on any `BaseComponentUpgradeable`, `AccessManager`, or `Router` contracts' child.

The `UPGRADER_ROLE` role can:

- Perform upgrades on any `BaseComponentUpgradeable`, `AccessManager`, or `Router` contracts' child.

The `AGENT_ADMIN_ROLE` role can:

- Enable/disable any agent.

The `SCANNER_ADMIN_ROLE` role can:

- Update any scanner at will.
- Register a new scanner under any address.
-

Enable/disable any scanner.

The `DISPATCHER_ROLE` role can:

- Link/unlink agents with scanners in the `Dispatch` contract.

The `SLASHER_ROLE` role can:

- Freeze a subject's total stake (including holders that staked on them) to prevent withdrawals.
- Slash a subject's total stake (including holders that staked on them).

The `SWEEPER_ROLE` role can:

- Withdraw any token that was sent to the `FortaStaking` contract, including extra `stakedToken` tokens.

The `REWARDS_ADMIN` role cannot perform any special action yet as any holder can give rewards to the protocol.

The `SCANNER_VERSION_ROLE` role can:

- Set the scanner node version.

The `MINTER_ROLE` role can:

- Freely mint new FORT tokens.

The `ADMIN_ROLE` role from the `FortaCommon` contract can:

- Add new addresses to the `MINTER_ROLE` role.
- Add new addresses to the `ADMIN_ROLE` role.
- Add new addresses to the `WHITELISTER_ROLE` role.
- Authorize upgrades in any `FortaCommon` contract child.
- Set the ENS reverse registration name for any `FortaCommon` contract child.

The `WHITELISTER_ROLE` role can:

- Add new addresses to the `WHITELISTER_ROLE` role.
- Add new addresses to the `WHITELIST_ROLE` role.

The `WHITELIST_ROLE` role can:

- Transfer un-vested FORT tokens freely to any other whitelisted address.
-

Receive un-vested FORT tokens.

- Deposit un-vested FORT tokens into a `StakingEscrow` contract.

Security model and trust assumptions

The current version of the protocol is strongly a fully centralized project, having multiple roles that could put in danger the users' funds: staked funds and un-vested tokens. Users must trust that the entities designated to take sensitive actions have the best interest in the protocol until a new revision comes into place. Furthermore, users must be whitelisted in order to "freely" transfer their assets, although it will only be possible to another whitelisted address.

The possible scenarios that could happen if one of the privileged addresses come into malicious hands could be:

- The imposibility to create a new `StakingEscrow` contract if the admin removes the `WHITELISTER_ROLE` from the `StakingEscrowFactory` contract.
- Upgrading the vesting wallets' functionality to allow the withdrawal of funds to a 3rd malicious actor.
- Disabling agents and scanners at will.
- Slashing any subject to send all those staked assets to a particular address stored in the `_treasury` variable.
- Lock every stakers' funds indefinitely.

That being said, the Forta team told us that there is a plan in place to replace the access controlled roles for with governance system that will handle these sensitive actions such as slashing, giving rewards, upgrading the contracts, and more, and become a decentralized project as it was meant to be.

Project design and maturity

Overall we are glad with the quality of the code, although documentation was lacking in most of the contracts. The protocol has plenty of moving pieces that are *mostly* connected between each other. Being an early iteration of the protocol, it has strong centralization of powers, which is to be expected from this early stage of the project. We feel that there are still a few links in the project that have not been addressed yet such as how to challenge or validate alerts on-chain, how to determine the severity of a malicious action, which rules will be applied to receive a reward, and the game design behind the incentives for nodes.

One note we have for the development of the protocol is that inheritance structures are complex, and the `__gap` variable in upgradeable contracts seems to have created problems for the developers. Thus,

we encourage a formal process for managing the `__gap` variables and documentation of the inheritance trees in the code.

Client-reported finding

During the audit, the client reported 1 vulnerability. Here we present the client-reported issue, followed by our findings.

Malicious user could dilute node's rewards

Holders can stake on behalf of a subject, and when they do, the proportion of the funds given by the node to themselves is diluted based on the amount of assets that holders staked. This brings the problem that because every single staked asset is treated as equal, when a subject receives rewards, the node may receive a small percentage of the rewards for its job, and malicious users may stake orders of magnitude higher than the node to take Between all the rewards from it, reducing the incentive to have nodes in the first place.

Critical severity

None.

High severity

[H01] Malicious user can register a scanner under any owner

The `ScannerRegistryCore` contract implements the functionality to allow the registration and minting of new scanners.

New scanners are meant to be registered by either calling the `register` function from the scanner's address or through the trusted forwarder, or by the admin when calling the `adminRegister` function. These functions then call the `_register` function which implements the rest of the registration.

However, this `_register` function is marked as a `public` function, meaning that any user could skip the checks and register a scanner in the same way as the admin does it.

In favor of restricting the admin functionalities to regular users, consider changing the visibility of the `_register` function to `internal`.

Medium severity

[M01] `__gap` missing in upgradeable contracts

The contracts `VestingWallet` and `VestingWalletV2` do not contain a `__gap` variable although they are upgradeable.

Consider adding a correct `__gap` variable to these contracts, or documenting a plan for managing storage collisions when upgrading the Vesting Wallet. Additionally, since upgradeable contracts with `__gaps` are used in many places within the contracts, consider implementing quality control steps for upgradeable contract development. For instance, make it a priority to check all `__gap` variables before pushing any new code commits, as well as leaving comments next to all variables in a contract indicating which storage slots they belong in. Consider leaving deprecated variables in the code, and leaving comments about the fact that they were deprecated to avoid confusion for future developers. Finally, consider implementing a predictable inheritance structure for all contracts and documenting it within each contract. Implementing these steps will reduce the surface for error and in the long run may save developer time by removing confusion about the storage layout of the contracts.

[M02] Lack of event emission after sensitive actions

The following functions do not emit relevant events after executing sensitive actions.

- The `sweep` function of the `FortaStaking` contract, after the `SWEEPER_ROLE` role withdraws all mistakenly sent tokens to the contract.

Consider emitting events after sensitive changes take place (including the first event emission in the constructor when appropriate), to facilitate tracking and notify off-chain clients following the contracts' activity

[M03] Unclear initialization of inherited contracts

The `StakeAwareUpgradeable` contract is inherited by a few contracts, such as the `ScannerRegistryEnable` contract, and therefore any other contract that inherits from those, such as the `ScannerRegistry` contract.

However, even though the `ScannerRegistry` contract implements the `initialize` function that initializes all the respective imports, there is no call to the `__StakeAwareUpgradeable_init` function from the `StakeAwareUpgradeable` contract. Furthermore, there is no single location in the whole codebase that would call the initializer to set the respective stake controller.

Moreover, the `FortaStaking` contract is inheriting the functionalities from the `ERC1155SupplyUpgradeable` contract but its `__ERC1155Supply_init` function is never initialized.

Consider calling all the respective initialization functions when inheriting functionalities from other contracts.

[M04] Lack of validation

Throughout the codebase, there are places where a proper input/output validation is lacking. In particular:

- In the `Router` contract, when adding or removing an element from the routing table, the methods return a boolean to inform the success of the call, but this output is never used or validated by the `Router` contract.
- Similarly to the case from above, in the `ScannerRegistryManaged` contract, when adding or removing a manager from storage, its method's output is never validated.
- In `Routed.sol`, the variable assignment in lines 13 and 25 are not validating if the address corresponds to a contract or if it is the zero address.
- In the `FortaStaking` contract it is possible to initiate a withdrawal and set in storage a deadline for a inexistent stake, emit several events during the process, and trigger an external hook.

A lack of validation on user-controlled parameters may result in erroneous or failing transactions that are difficult to debug. To avoid this, consider adding input and output validation to address the concerns raised above and in any other place when ### Lack of validation

Throughout the codebase, there are places where a proper input/output validation is lacking. In particular:

- In the `Router` contract, when adding or removing an element from the routing table, the methods return a boolean to inform the success of the call, but this output is never used or validated by the `Router` contract.
- Similarly to the case from above, in the `ScannerRegistryManaged` contract, when adding or removing a manager from storage, its method's output is never validated.
- In `Routed.sol`, the variable assignment in lines 13 and 25 are not validating if the address corresponds to a contract or if it is the zero address.
- In the `FortaStaking` contract it is possible to initiate a withdrawal and set in storage a deadline for a inexistent stake, emit several events during the process, and trigger an external hook.

A lack of validation on user-controlled parameters may result in erroneous or failing transactions that are difficult to debug. To avoid this, consider adding input and output validation to address the concerns raised above and in any other place when appropriate.

[M05] Staked funds might get soft-stuck

The `FortaStaking` contract implements the functionality to allow holders to stake their funds (vested or not) into a subject and collect rewards by doing so. The contract uses 2 different accounting systems to handle the assets: a `Distribution` type based for the asset in stake units and the inner ERC1155 accounting system for the associated shares. When a user stakes, the contract mints new active shares. When a user wants to withdraw, the contract burns those active shares and mints inactive ones.

When minting these ERC1155, the `_doSafeTransferAcceptanceCheck` hook will get triggered and it would check if the destinatary is a `ERC1155Receiver` implementer or not when it detects that the address has code in it.

However, if the minting process happens during the constructor of a non-fully compatible ERC1155 wallet, the `FortaStaking` contract would treat the destinatary as a regular EOA during the minting process. Then, once it is deployed, the wallet will not be able to mint new shares due to the same `_doSafeTransferAcceptanceCheck` hook as it will get triggered, failing at the same validation that was skipped on the first deposit.

This means that when the wallet starts the process to withdraw the assets, the transaction will fail during the minting of inactive shares. Nevertheless, the user may transfer those active shares to a fully compatible wallet to then initiate the withdrawal process once again.

In favor of improving the usability of the protocol, consider documenting the requirements that 3rd party wallets would need to be able to fully interact with it either during the deployment stage or once it has been deployed.

[M06] L2 tokens could get stuck

The `release` function in the `StakingEscrow` contract allows users to send tokens from the `StakingEscrow` contract to another account on L2. However, in order to preserve the vesting schedule, FORT tokens are NOT allowed to be transferred unless they have been accounted for by `pendingReward`.

Typically if tokens are received as rewards from the staking contract, `pendingReward` will be increased. However, if tokens are sent directly to this contract, it will not increase `pendingReward` and the tokens will not be transferable. Instead, users will only be able to `bridge` their FORT tokens, subjecting them to the vesting schedule on L1.

Consider implementing some accounting to allow for users to transfer FORT tokens to this contract, and afterwards transfer them out via the `release` function. Alternatively, consider making a clear warning to any `StakingEscrow` contract users that FORT tokens transferred to the contract will not be `release`-able and will be subjected to vesting.

[M07] Slashing process could be reverted

When a certain subject under-performed or has done actions against the correct operation of the protocol, the `SLASHER_ROLE` role can slash that subject and all the users that have staked on it by calling the `slash` function from the `FortaStaking` contract. After the value that should be taken from inactive and active stake is computed, the slashed funds are transferred to the `_treasury` address.

However, if the `_treasury` address is being set as zero either during the initialization of the contract or by the `DEFAULT_ADMIN_ROLE` role with the `setTreasury` function, the whole slashing mechanism will not work because the `FORT` token does not allow to transfer tokens to the zero address.

In order to prevent the possible reversion of the slashing process, consider always validating that the `_treasury` address is not zero when initializing the contract or when a new treasury address is being set.

Low severity

[L01] TODOs and comments implying unfinished code

There are "TODO" comments and other comments implying unfinished codes in the codebase. These should be tracked in the project's issues backlog. In particular:

- Line 38 of `AgentRegistryCore.sol`.
- Line 54 of `AgentRegistryCore.sol`.
- Line 35 of `Router.sol`, which seems to imply the development effort here is unfinished.

During development, having well described "TODO" comments will make the process of tracking and solving them easier. Without that information, these comments might tend to rot and important information for the security of the system might be forgotten by the time it is released to production.

These TODO comments should at least have a brief description of the task pending to do, and a link to the corresponding issue in the project repository.

Consider updating these comments to add this information. For completeness and traceability, a signature and a timestamp can be added.

[L02] Add information in `_emitHook` calls

Within the `FortaStaking` contract, there are many calls to the `_emitHook` function for the hook `hook_afterStakeChanged`. However, this hook only includes two parameters: `subjectType` and `subject`. Note that identical calls are made in the `deposit`, `initiateWithdrawal`, `withdraw`, and the `slash` functions.

If it is eventually needed to determine which user triggered a stake change, which user's stake has changed, by how much a stake has changed, or what function resulted in the stake change, a call into the `FortaStaking` contract will be needed, possibly alongside complex logic in an external contract.

Consider passing relevant data with the calls to `_emitHook`, such as `_msgSender` and `changeInStake`. Doing so will make it easier for the contract recieving the hook to interpret what has happened. Additionally, consider documenting the purposes of the hooks for future development, so it is clear exactly which data may be needed from the `hook_afterStakeChanged` call.

[L03] Inconsistent slot size for upgrades

Throughout the whole codebase, several contracts are allowed to be upgradable in order to improve/ extend the functionalities or to fix a vulnerability. To mitigate the possibility of having a storage collision, those contracts define an array at the bottom of the contract that its length added to the number of variables defined in the contract adds to a fix number, usually 50.

However, there are contracts in which the sum is not consistent with the rest of the codebase. In particular:

- The `ScannerRegistryManaged` contract whose sum adds up to 45.
- The `StakeAwareUpgradeable` contract whose sum adds up to 5.
- The `AgentRegistryCore` contract whose sum adds up to 45.

In order to improve the code's readability, prevent future storage collisions on contracts that may have less storage slots available, and to be consistent with the rest of the code, consider fixing all the respective places where the sum does not add up to a common fixed number. Furthermore, consider documenting as in-line comments all the variables that took one of those places as documentation for the `__gap` variable as an exercise to double corroborate its final length.

[L04] Potential for hash collisions with frontrun protection

Within `AgentRegistryCore.sol`, calls to the `frontrunProtected` modifier utilize `abi.encodePacked` to create a "unique" hash of some commited data.

However, by using two dynamic parameters next to each other (both in `createAgent` and in `updateAgent`), hashes can be easily forged simply by adjusting `metadata` and `chainIds`, such that an agent may be created or updated incorrectly.

By simply using `abi.encode` rather than `abi.encodePacked`, such collisions from dynamic parameters being adjacent can be avoided. Consider using `abi.encode` here instead.

[L05] Implement a remove-whitelist functionality

Currently, to prevent tokens from being transferred while within the vesting schedule, a `WHITELIST_ROLE` is defined and checked in `FortaCommon._beforeTokenTransfer`. The user or contract can only transfer tokens if "whitelisted". Eventually, the whitelist requirement is planned to be removed via a contract upgrade.

Since contract upgrades are notoriously risky due to storage slot assignment issues, and since the removal of the whitelist functionality is planned for future development, consider implementing an access controlled "remove whitelist" functionality in the existing contract which disables the whitelist. This will also give users greater confidence about the future state of the system and allow development which may depend on the future system design to proceed more smoothly.

[L06] Implicit casting

Throughout the codebase, an instance of implicit casting between types has been detected.

In the `FortaStakingUtils` library, in lines 10 and 18, the `subjectType` `uint8` parameter is being used in a bitwise `OR` operation against a `uint256` result.

Whenever a different type of variable is needed, consider either checking and casting the variable into the desired type or using OpenZeppelin's `SafeCast` library which provides overflow checking when casting from one type of number to another.

[L07] Incomplete interfaces

The `IRouter` interface should have an externally accessed function which is not being declared in the interface, the `version` getter function.

Consider declaring all externally accessed functions without access control so users and developers can make use of interfaces when using the protocol.

[L08] Disabled scanners and agents may appear to be linked

Within `Dispatch.sol`, the mappings `scannerToAgents` and `agentToScanners` should store correct linkages between agents and scanners.

If an agent or scanner is disabled, they are not allowed to be `linked`. Thus, it follows that if scanners or agents which are `linked` are then disabled, the `link` should be removed. However, this is not enforced - a scanner or agent may be disabled, but the values in `scannerToAgents` and `agentsToScanners` may not reflect this.

Consider adding a programmatical way to remove `link`s whenever an agent or scanner is disabled. Consider that, since a single agent or scanner may have multiple instances it is linked to, the `unlink`ing process may involve calling `unlink` multiple times. This may mean adding a limit to the number of links for a single instance. Alternatively, consider documenting this behavior clearly for any 3rd-party developers, and encouraging them to double-check that both the scanner and agent are enabled when querying linked pairs.

[L09] Magic numbers are used

Throughout the codebase, there are occurrences of literal values with unexplained meaning. For example, the operation to get the maximum slashable stake in the `FortaStaking` contract uses explicit numbers during the calculation without documenting the reasons of such values.

To improve the codeâ€™s readability and facilitate refactoring, consider defining a constant for every magic number, giving it a clear and self-explanatory name. For complex values, consider adding an inline comment explaining how they were calculated or why they were chosen.

[L10] Non-registered scanners default to non-disabled states

The `ScannerRegistryEnable` contract implements the functionality for enabling and disabling scanners, among other actions, and it uses bit maps to keep track of their states.

However, when enabling a scanner, the `_scannerEnable` internal function uses the inverse assignment, meaning that a `1` flag accounts as a disabled state and a `0` state as an enabled one. Although this will not be sufficient to mark a random scanner ID as enabled, as it must be registered as a ERC721 and have enough stake on top of it, its default un-registered behavior resembles a non-disabled scanner.

Even though this does not possess a security risk per se, in order to improve the readability of the code and reduce the attack surface, consider using a different state from the default one when enabling new scanners.

[L11] Semantic overload

The `ScannerRegistryEnable` contract implements the functionality for enabling and disabling scanners, but it also extends the functionality of the registration process.

When registering a new scanner, the contract checks if the minimum stake for the scanner type is greater than zero. This value is changed by the admin in the `FortaStaking` contract and it is meant to define a threshold value instead of an enabled status.

This is known as Semantic Overload. If the multiple meanings of the variables and states are not totally clear when making changes to the code, it can introduce severe vulnerabilities. We strongly discourage its usage if possible.

Consider explicitly setting independent flags to represent the state of the scanners instead of using the same variable for different purposes.

[L12] Deviation from specifications

The `StakingEscrow` contract implements the functionality to allow users who have vested tokens in L1 to be able to interact, participate, and stake those assets in the protocol. By doing so, the protocol could disburse rewards to users which would be collectable by calling the `release` function, where the documentation states that even if these assets are sent to a non-whitelisted address, the tokens will arrive but those will get stuck.

However, due to the `_beforeTokenTransfer` function hook being called during any regular transfer of ERC20 tokens, it will not be possible to send those assets to a non-whitelisted address.

Consider either updating the documentation to reflect the current behavior of the protocol or fixing the implementation to follow the specifications.

[L13] Reentrancy possibility due to `_doSafeTransferAcceptanceCheck`

After calling the `_mint` function in the `FortaStaking` contract, the function `_doSafeTransferAcceptanceCheck` from the `ERC1155Upgradeable` contract will get called, which will hand over control to the `to` address.

This does not happen similarly for the `_burn` function because the `to` address would be the `address(0)`. In the `FortaStaking` contract, the `deposit` and `initiateWithdrawal` functions both make a call to the `_mint` function followed by a call to the `emitHook` function. The `emitHook` calls are currently undefined for these functions, but if they rely on calling into the Forta contracts to access certain variables, for example, then a malicious user may be able to affect these variables in unexpected ways.

When developing Forta, specifically the "hook" calls, developers should consider the fact that arbitrary code may be executed in the frame of the `_mint` function call. Since more control exists for the hooks, consider moving the call to the `_emitHook` function prior to the `_mint` function call in these indicated cases. Additionally, consider forwarding relevant data in the `_emitHook` function calls as it is needed (such as balances or ERC1155 tokens, or total supply) so that if they are tampered with in the context of the `_mint` function call they will not affect the `_emitHook` function call.

Notes & Additional Information

[N01] Add bridge exit instructions

Currently in the codebase, there is no obvious way to release tokens from the Polygon POS bridge. According to their documentation, the `exit` function will need to be called on Ethereum to release tokens sent from Polygon to Ethereum.

This may not be clear to users who transfer tokens from L2 to L1. The only locations in the code referencing the `exit` function are within the `IRootChainManager` interface and within the mock for `RootChainManager`. Notably, in the mock, the function is not implemented. It appears that the "exit" functionality may not have been fully implemented as intended.

Consider implementing instructions for users to "exit" from the Polygon bridge. To make it easier for users, consider calling the function from within contracts they are already interacting with, and ensure that instructions for using these functions are clear. Make sure to test all exit functionality thoroughly. Alternatively, consider writing a guide for users to exit on their own, and include it within the Forta documentation. Tests for such instructions should also be performed.

[N02] Suggestion: add hooks which revert on failure

Currently in the codebase, the `Router` and `Routed` contracts allow for adding hooks via the `_emitHook` function to different contracts throughout the Forta codebase. Notably, different target contracts can be added and changed for each hook at any time, and all hooks will not revert the overall call if they revert.

It appears that this architecture exists for handling an unplanned future architecture. Therefore, consider also adding functionality which allows for hooks which MUST succeed, or revert the outer call if they revert. Note that this suggestion is merely for better future development experience, and does not arise from any security issue.

[N03] Add explicit warning about the Forwarder

The `Forwarder` contract allows for users to sign messages, and for them to be executed by other EOA's by presenting a valid signature. It includes nonces for replay protection for such transactions. However, users should be made explicitly aware that failing transactions will NOT consume a nonce. This is because the `_verifyAndConsumeNonce` function cannot store any data within a reverting transaction. Meaning, any transactions which revert can be broadcast again at any point in the future, by any user, and may succeed at that time.

Although this is briefly mentioned in the linked README, consider adding an explicit inline warning that nonces are not consumed in failing transactions, unlike normal ethereum transactions. Users should be made aware that to remedy this, they should make use of the deadline functionality, or that they will need to successfully broadcast a transaction which succeeds with the same nonce to cancel another.

Additionally, consider adding a `cancelTransaction` function, which simply validates a signature and consumes the nonce.

[N04] Gas optimizations

There are a few spots identified in the code which could be optimized for gas consumption.

- On line 372 of `FortaStaking.sol`, a call to the `availableReward` function is made. This calls the `subjectToActive` method. On line 373, another call to the `subjectToActive` method is made. Making a single call to `subjectToActive` to utilize in both locations could save gas.
- In `StakingEscrowFactory.sol`, the `immutable` value `template` is declared as an instance of the `StakingEscrow` contract. However, it is only ever used in the codebase when it is casted to an `address`. Instead of storing it as type `StaingEscrow`, store it as type `address` to save gas during the deployment and the runtime execution.

Consider correcting these two instances for more efficient gas usage when interacting with the Forta codebase.

[N05] Inconsistent format in error messages

Error messages throughout the code base were found to be following different formats. In particular, some messages are formatted "Contract name::function name: error message", whereas others are not.

So as to favor readability and ease debugging, consider always following a consistent format in error messages and, furthermore, consider adapting all revert messages to the `"Contract name::function name: error message"` format.

[N06] Misleading documentation

Throughout the codebase, there are places of misleading documentation. In particular:

- The `FortaBridgedPolygon` contract implements the functionality to handle the bridge of asset between the 2 networks. When depositing funds in the root chain, the `childChainManagerProxy` address will call the `deposit` function to mint the respective tokens in the child chain. The in-line documentation also states that "To avoid token locked on the parent chains not being correctly represented on the child chain, this should NEVER revert. Consequently, we might have to temporarily grant WHITELIST_ROLE to the receiver", however the `_mint` function can revert if the `_maxSupply` value is reached.
- In the `Router` contract, the in-line documentation states that the contract "should be BaseComponentUpgradeable, because BaseComponentUpgradeable is Routed" where it should say "shouldn't be" instead.
-

> In the `FortaStaking` contract, the `sweep` function allows to withdraw any token that was mistakenly sent to the contract. However, the documentation suggest that the function sweeps all the tokens at the same time, even though the function takes one at a time.

Consider fixing the documentation so users are aware of the real behavior of the protocol.

[N07] Missing Docstrings

Many of the contracts and functions in the Forta Token codebase lack documentation. This hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security, but also correctness. Additionally, docstrings improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned and the events emitted.

Consider thoroughly documenting all functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

[N08] Naming issues

There are many areas in the codebase which we feel better naming could greatly benefit development and reviewer's understanding of the code. In particular:

- Within `Forwarder.sol`, there are two functions named `getNonce`. Consider renaming one or both of them to indicate the difference between the two functions.
- Within `Dispatch.sol`, `function agentsFor` should be renamed to `numAgentsFor` since it returns the number of agents for a scanner. Similarly, `function scannersFor` should be renamed to `numScannersFor`.
- Within `Dispatch.sol`, `function agentsAt` should be renamed to `agentAt` since it returns only a single agent. Similarly, `function scannersAt` should be renamed to `scannerAt`.
- Within `VestingWalletV2.sol`, `function setHistoricalBalanceBridged` should be renamed to `setHistoricalBalanceMin` to better represent what it does. Similarly, `function updateHistoricalBalanceBridged` should be renamed to `updateHistoricalBalanceMin`.

Note also that in a previous round of auditing, the issue was reported that many upgradeable contracts had filenames which did not indicate upgradeability. Consider making the suggested naming changes to better explain the code's purpose and reduce confusion for reviewers and developers.

[N09] Lack of `_commits` getter

The `FrontRunningProtection` contract utilizes a mapping, `_commits`, to track hashes for a commit-reveal scheme.

However, there is no easy way to access the `_commits` mapping on-chain. Consider adding a getter for the `_commits` mapping to ease the user and developer experience. Alternatively, if there is a reason for keeping the `_commits` mapping as `private`, consider explaining it in a comment.

[N10] Pragma statement is not consistent

Although most of the contracts in the codebase use a `pragma` statement of `^0.8.0`, the `index.sol` file uses a `>=0.8.4` version instead.

Although this does not represent a security risk per se, it is always recommended to use the same `pragma` statement for all the codebase.

Consider reviewing and updating the `pragma` statements of all contracts throughout the code base to ensure they are consistent.

[N11] Unneeded `public` visibility

Some functions in the codebase have `public` visibility, although it is unneeded since they are not called within the contract they exist in. For example:

- Both `getNonce` functions within `Forwarder.sol`.
- The `execute` function within `Forwarder.sol`.
- The `mint` function within `Forta.sol`.

Consider changing the visibility of these functions to `external` to better indicate their role in the codebase, and to follow Solidity best practices. Alternatively, if the functions are needed to be public, consider documenting this with a comment.

[N12] Inconsistent use of roles

Throughout the codebase, there are several roles in charge of performing unique and sensitive actions. Most of such roles are defined in the `Roles.sol` file, one of them being the `ENS_MANAGER_ROLE` role which is in charge of setting the ENS reverse registration.

However, in the `FortaCommon` contract, this task belongs to the `ADMIN_ROLE` role instead.

Consider either using the `ENS_MANAGER_ROLE` role instead of the `ADMIN_ROLE` role to set such ENS variable or documenting the reason to use the `ADMIN_ROLE` role to improve the readability of the code and reduce the attack surface.

[N13] Multiple contracts per file

The file `Forwarder.sol` contains two contracts: `EIP712WithNonce` and `Forwarder`.

Consider separating the contracts into their own files to make the codebase easier to understand for developers and reviewers.

[N14] Styling issues

Within the codebase, we found a few stylistic issues which, if corrected, would make the codebase easier to review and more understandable, as well as more predictable for future development efforts. Below are our findings:

- There is an extra line at `ScannerRegistry.sol` line 5.
- There is an extra line at `ScannerNodeVersion.sol` line 33.
- The `_setStakeController` `private` function is placed before `public` functions in `StakeAware.sol`.
- Within the `initialize` function of `VestingWallet.sol`, input parameter names contain a trailing underscore, unlike other `initialize` functions in the codebase which use two leading underscores for their input parameter names.
- Throughout the codebase, there is relatively common usage of the expression `++i` in places where `i++` would function identically. For example, within a `for` loop in `AgentRegistryCore.sol`. This is in contrast to `i++` which is also frequently used, for instance in the `for` loop in `BatchRelayer.sol`. Consider using only one style whenever possible, and when necessary to use the other style, including an explanatory comment.
- In `StakingEscrow.sol`, there are two `deposit` functions (one for a "full" deposit and one for a partial deposit). However, the `initiateWithdrawal` function and its partner, `initiateFullWithdrawal` do not match this pattern. Consider overloading the `initiateWithdrawal` function as well, to match the style of the overloaded `deposit` function.

[N15] Typos and erroneous comments

Several typographical errors were found in the codebase which should be corrected. In particular:

- Line 26 of `FullMath.sol` says "Remiander" instead of "Remainder".
-

Line 29 of `FortaStaking.sol` says "elligible" instead of "eligible".

- Line 401 of `ForaStaking.sol` says "cal" instead of "can".
- Line 49 of `StakingEscrow.sol` mentions `__l1vesting` in the error message, instead of `__l2manager`.
- Line 112 of `StakingEscrow.sol` references the "beneficiary", but should instead say "manager".
- Line 67 of `VestingWalletV2.sol` says "explicitelly" instead of "explicitly".

Consider correcting the listed typos and errors within comments. Consider utilizing a spell-checker, such as codespell for future changes to the codebase.

[N16] Unneeded or unclear frontrunning protection to update an agent

The `FrontRunningProtection` contract implement the functionality to allow a commit-reveal scheme to run frontrunning-protected actions in the protocol. Between those, are the ones to create and update an agent in the `AgentRegistryCore` contract.

Although it may be necessary to have such scheme to create a new agent, so no one can own it beforehand, it is not clear why would it be necessary to have the same procedure to update an agent, taking into account that only the owner of such agent is able to perform the update.

In order to improve the readability of the code and the UX of the protocol, consider documenting the reasoning behind the need of the commit-reveal scheme during agent updates.

[N17] Vesting schedule is changeable

When using the VestingWallet or VestingWalletV2 contracts, users should note that their `vestedAmount` value is dependent on `_historicalBalance` output, which may change. If they transfer tokens to the `VestingWallet` instance, they will increase the `_historicalBalance` output which will result in a changed vesting schedule. Specifically, it will treat the new token amount as vested as well, and it will be released to the user following the vesting schedule. This has the effect of locking up some of the users funds temporarily if they accidentally transfer them to the `VestingWallet` contract.

This is made more complicated by the fact that the staking system is set up to only allow funds to be transferred back to the `VestingWallet` contract. So, users must transfer their tokens back to the `VestingWallet` contract in order to be able to transfer them freely, and at that point they will be re-vested.

Furthermore, any change in the staked value in L2, e.g. funds get slashed, will not be addressed in L1 due to the `historicalBalanceMin` variable, off-syncing the balances between both chains even if all the remaining staked funds in L2 are bridged back to L1.

Users should also be made aware of the existence of the `setHistoricalBalanceBridged` function and the `updateHistoricalBalanceBridged` function, which give the `owner` role power to instantly affect the vesting schedule, specifically changing its speed.

Consider refactoring the vesting schedule logic to make it more predictable and user-friendly. For example, consider tracking the remaining vesting balance independently from the balance of the contract, so that tokens which are transferred to the `VestingWallet` contract from L2 are not re-vested. Additionally, consider defining predictable contract logic for calling the `setHistoricalBalanceBridged` and `updateHistoricalBalanceBridged` functions so that users can be assured their vesting schedules will not be tampered with maliciously. Finally, consider the legal implications of changing a vesting schedule based on both user and admin actions, as well as the tax implications for a user whose vesting schedule may change without warning.

Conclusions

No critical issues and one high severity issue were found. Some changes were proposed to follow best practices and reduce the potential attack surface.