The Graph - PR-548, 549

Date	March 2022
Auditors	Martin Ortner, Elias Leers

1 Executive Summary

This report presents the results of our engagement with **The Graph** to review **PR-548** and **PR-549** in their contracts repository. We conducted this assessment over two weeks, from **Mar 14–25, 2021**, and allocated 2x2 person-weeks.

At first, we focused on **PR-548** implementing gip-0024: Query Versioning for the `DisputeManager`, laying the tracks to allow the registration of disputes using the new semantic versioning scheme. The changeset re-uses `DisputeManager`'s hardcoded `DOMAIN_SEPARATOR` `version` field and refactors it to allow a user-provided domain separator to be passed in instead of the formerly hardcoded one. This was quickly communicated to the client as a potential degradation of the security of the system. However, as noted with the finding accordingly, the client stated that the on-chain domain separator verification (`chainId`, `address(DisputeManager)`) is obsolete because the off-chain arbitrators now carry the burden of verifying the signing domain.

The second focus was **PR-549** which implements 'altruistic signaling'. The core change is that this pull request will allow zero token allocations. Zero token allocations signal that someone supports a subgraph and provides indexing services without receiving rewards for it. Altruistic indexers still have to provide a bond that can be slashed if they do not comply with the protocol. In addition, this implementation allows anyone to close stale allocations after the max allocation period, but only the indexer itself can close their altruistic allocation. A seemingly minor change included with this PR is that `EpochManager.currentEpoch()` now returns `old_epoch + 1`, making epochs start at `1` instead of `0`. According to the client, the currently deployed `EpochManager` will not be upgraded. This change should be seen as a preparation for the next deployment (updating the existing manager would cause one epoch to be skipped). The PR also slightly changes the Allocation Lifecycle/State diagram. `AllocationState.Claimed` was formerly indicated by `alloc.tokens == 0` in `_getAllocationState()` and is now indicated by `alloc.createdAtEpoch == 0`.

- PR-548 - see Section 3 - PR-548
- PR-549 - no security relevant issues found.

In addition, we also clarified whether an indexer/operator can always choose `_updateRewards` over `distributeRewards` by not providing a `poi` with their call. The question was whether this means they can block delegator rewards distribution indefinitely.

The client provided the following statement:

> When they present a `POI = 0x0` they are opting out of indexing rewards, this might happen because they can't generate it due to software bugs, or any other reason. Effectively delegators won't receive rewards as a consequence. If this happens repeatedly by the same indexers, delegators might decide to undelegate from them due to bad performance.

2 Scope

Our review focused on the following Pull-Requests:

- PR-548 @ a48c42d0ece39002df33ddad31c894e3213923d6
- PR-549 @ 9385b40faedd77c9af14eb6703b9c3f459e30949

3 PR-548

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.

3.1 DisputeManager - PR-548 removes replay protection Major ✓ Works as Designed

Resolution

According to the client this is works as designed. Anyone/Fishermen can submit disputes. The smart contract currently accepts signatures for domain separators that belong to different Dispute managers or are signed for other chains. The burden of validating that a dispute is valid for the expected smart contract and chainId is offloaded to the Arbitrators. They have to validate the disputes (including whether they are valid for the current chain) and take action on them by calling `acceptDispute`, `rejectDispute`, or `drawDispute`. The arbitrator decision logic was not in scope for this review.

The client provided the following statements:

> [...] The fisherman is the entity that presents the Dispute (it can be really anyone), but the Arbitrators are the one deciding if a dispute is valid from the ones submitted to the contract. So a dispute for an invalid version or with an invalid domain separator will be slashed by the Arbitrators as defined by the Arbitration Charter https://hackmd.io/@4Ln8SAS4RX-505bIHZTeRw/BJcHzpHDu

and

> [...] That means it's not really important to have the contract fix the domain separator for this use case and validate it onchain on submission, we use it to ensure that we are attesting to the right structure and the Arbitrators will decide if to resolve the dispute if everything looks right

Description

PR-548 introduced a subtle change to the structure of the encoded hash receipts that effectively disables replay protection for signed hash receipts. Note that signed hash receipts are the basis for opening and resolving disputes and can be used to slash indexers.

Dissecting the ChangeSet

The relevant changes to `DisputeManager` are as follows:

- **(1)** The hardcoded **DOMAIN_SEPARATOR** is removed from the contract.

```
diff --git a/contracts/disputes/DisputeManager.sol b/contracts/disputes/DisputeManager.sol
index de998ad..abf4665 100644
--- a/contracts/disputes/DisputeManager.sol
+++ b/contracts/disputes/DisputeManager.sol
@@ -41,33 +41,31 @@ contract DisputeManager is DisputeManagerV1Storage, GraphUpgradeable, IDisputeMa

 // -- EIP-712  --

-    bytes32 private constant DOMAIN_TYPE_HASH =
-        keccak256(
-            "EIP712Domain(string name,string version,uint256 chainId,address verifyingContract,bytes32 salt)"
-        );
-    bytes32 private constant DOMAIN_NAME_HASH = keccak256("Graph Protocol");
-    bytes32 private constant DOMAIN_VERSION_HASH = keccak256("0");
-    bytes32 private constant DOMAIN_SALT =
-        0xa070ffb1cd7409649bf77822cce74495468e06dbfaef09556838bf188679b9c2;
 bytes32 private constant RECEIPT_TYPE_HASH =
     keccak256("Receipt(bytes32 requestCID,bytes32 responseCID,bytes32 subgraphDeploymentID)");
```

Here is where the **DOMAIN_SEPARATOR** was constructed on deployment. Note that this hash must be constructed at deploy time because the required information (`chainID` and `address(this)`) is not known before. It should then be stored to an **immutable** state var to guarantee that this domain hash will not change.

```
@@ -179,18 +177,6 @@ contract DisputeManager is DisputeManagerV1Storage, GraphUpgradeable, IDisputeMa
         _setMinimumDeposit(_minimumDeposit);
         _setFishermanRewardPercentage(_fishermanRewardPercentage);
         _setSlashingPercentage(_qrySlashingPercentage, _idxSlashingPercentage);
-
-        // EIP-712 domain separator
-        DOMAIN_SEPARATOR = keccak256(
-            abi.encode(
-                DOMAIN_TYPE_HASH,
-                DOMAIN_NAME_HASH,
-                DOMAIN_VERSION_HASH,
-                _getChainID(),
-                address(this),
-                DOMAIN_SALT
-            )
-        );
     }
```

- **(2)** `encodeHashReceipt` now takes an argument `_domainSeparator`

What formerly used to be a hardcoded **DOMAIN_SEPARATOR** is now variable and passed to the function as an argument. The important question here is: Will the domain separator be provided by an untrusted party, and how does the contract ensure that it does not accept messages signed for a foreign domain.

```
     /**
@@ -299,14 +285,20 @@ contract DisputeManager is DisputeManagerV1Storage, GraphUpgradeable, IDisputeMa
      * https://github.com/ethereum/EIPs/blob/master/EIPS/eip-712.md#specification.
      * @notice Return the message hash used to sign the receipt
      * @param _receipt Receipt returned by indexer and submitted by fisherman
+     * @param _domainSeparator EIP-712 domain separator
      * @return Message hash used to sign the receipt
      */
-    function encodeHashReceipt(Receipt memory _receipt) public view override returns (bytes32) {
+    function encodeHashReceipt(Receipt memory _receipt, bytes32 _domainSeparator)
+        public
+        pure
+        override
+        returns (bytes32)
+    {
         return
             keccak256(
                 abi.encodePacked(
                     "\x19\x01", // EIP-191 encoding pad, EIP-712 version 1
-                    DOMAIN_SEPARATOR,
+                    _domainSeparator,
                     keccak256(
                         abi.encode(
                             RECEIPT_TYPE_HASH,
```

- **(3)** `_recoverAttestationSigner` takes a parsed `Attestation` struct and constructs the message hash for it using the `domainSeparator` that is stored in the `_attestation`. The important question here, again, is: Where does the `domainSeparator` come from, is the domain verified in the context of this contract, and can it be trusted?

```
@@ -729,7 +722,7 @@ contract DisputeManager is DisputeManagerV1Storage, GraphUpgradeable, IDisputeMa
  */
 function _recoverAttestationSigner(Attestation memory _attestation)
     private
-        view
+        pure
     returns (address)
 {
     // Obtain the hash of the fully-encoded message, per EIP-712 encoding
@@ -738,7 +731,7 @@ contract DisputeManager is DisputeManagerV1Storage, GraphUpgradeable, IDisputeMa
         _attestation.responseCID,
         _attestation.subgraphDeploymentID
     );
-        bytes32 messageHash = encodeHashReceipt(receipt);
+        bytes32 messageHash = encodeHashReceipt(receipt, _attestation.domainSeparator);

     // Obtain the signer of the fully-encoded EIP-712 message hash
     // NOTE: The signer of the attestation is the indexer that served the request
@@ -749,25 +742,13 @@ contract DisputeManager is DisputeManagerV1Storage, GraphUpgradeable, IDisputeMa
         );
 }
```

- **(4)** `_parseAttestation` parses a `bytes` array into an `Attestation` struct. We can see that the **DOMAIN_SEPARATOR** is part of presumably untrusted input.

Note that the signature parameters were created for message hash that is only correct for the provided `attestation.domain_separator` and that `encodeHashReceipt` uses this user-provided **DOMAIN_SEPARATOR** to rebuild the message hash. **There is no check whether the message to be verified was signed for the contracts domain in the first place**. This might allow an attacker to provide a validly signed message for/from a different contract/chain to be validated correctly on this contract, which can be problematic.

```
     /**
      * @dev Parse the bytes attestation into a struct from `_data`.
      * @return Attestation struct
      */
     function _parseAttestation(bytes memory _data) private pure returns (Attestation memory) {
         // Check attestation data length
-        require(_data.length == ATTESTATION_SIZE_BYTES, "Attestation must be 161 bytes long");
+        require(_data.length == ATTESTATION_LENGTH, "Attestation must be 161 bytes long");

         // Decode receipt
         (bytes32 requestCID, bytes32 responseCID, bytes32 subgraphDeploymentID) = abi.decode(
@@ -781,7 +762,10 @@ contract DisputeManager is DisputeManagerV1Storage, GraphUpgradeable, IDisputeMa
         bytes32 s = _toBytes32(_data, SIG_S_OFFSET);
         uint8 v = _toUint8(_data, SIG_V_OFFSET);

-        return Attestation(requestCID, responseCID, subgraphDeploymentID, r, s, v);
+        // Decode domain separator
+        bytes32 domainSeparator = _toBytes32(_data, DOMAIN_SEPARATOR_OFFSET);
+
+        return Attestation(requestCID, responseCID, subgraphDeploymentID, r, s, v, domainSeparator);
     }
```

Decomposing the signed structure data

So far, we have learned that receipts are signed in EIP-712: Ethereum typed structured data hashing and signing format, with EIP-191 padding, a **DOMAIN_SEPARATOR**, and payload that is prefixed with the `PAYLOAD_TYPE_HASH`.

```
keccak256(
                abi.encodePacked(
                    "\x19\x01", // EIP-191 encoding pad, EIP-712 version 1
                    DOMAIN_SEPARATOR,
                    keccak256(
                        abi.encode(
                            RECEIPT_TYPE_HASH,  /* PAYLOAD TYPE HASH */
                            _receipt.requestCID,      /* PAYLOAD ... */
                            _receipt.responseCID,
                            _receipt.subgraphDeploymentID
                        ) // EIP 712-encoded message hash
                    )
                )
            );
```

- **(1)** (`"\x19\x01"`)

Indicates that this is an EIP-191 typed message (`\x19`) and the following data is EIP-712 "Structured data" (`\x01`).

- **(2) DOMAIN_SEPARATOR**

Serves the purpose of **pinning** the signed data to a specific domain. EIP-712 suggests the following domain fields:

```
EIP712Domain(string name,string version,uint256 chainId,address verifyingContract,bytes32 salt)
```

Using all of the fields would pin the signature to:

  - string name the user readable name of signing domain, i.e. the name of the DApp or the protocol.
  - string version the current major version of the signing domain. Signatures from different versions are not compatible.
  - uint256 chainId the EIP-155 chain id. The user-agent should refuse signing if it does not match the currently active chain.
  - address verifyingContract the address of the contract that will verify the signature. The user-agent may do contract specific phishing prevention.
  - bytes32 salt an disambiguating salt for the protocol. This can be used as a domain separator of last resort.

- **(3)** (Payload)

Arbitrary EIP-712 typed payload data. This is the actual data that is being signed and/or verified.

Security Implications

The reason why eth signed data contains a **DOMAIN_SEPARATOR** is to ensure that the data can only be used with this domain. On smart contract side this is usually enforced by a hardcoded domain separator generated when the contract is constructed. It typically includes the `chainId` the smart contract was deployed on and the contract's address. If a message is to be verified, the smart contract verification routine hashes the user's payload and mixes this with the hardcoded domain separator that was generated when the contract was deployed. This way it is implicitly enforced that messages must be signed for this smart contract (as `address(this)` is part of the domain) and that the `chainId` used in the signed data matches the one of this contract. Messages that are signed for a different domain (i.e., messages from a different chain or messages signed for a different contract address) are automatically rejected as the signature parameters do not match the message hash constructed with the verifying contract.

Recommendation

We understand there is a need to add an application version scheme to the signed data. It is recommended to distinguish between application versions (more complex behavior) and the version of the signing domain. Therefore, it is recommended, to add the application version to the EIP-712 signature payload, keeping the `DOMAIN_SEPARATOR` as is, and instead updating the `RECEIPT_TYPE_HASH` to also include the semantic version information. This way, it is ensured that `chainId` and `address(this)` must match for every signature (as they are hardcoded in the `DOMAIN_SEPARATOR` already). Hence, messages from other chains or messages that are signed for different `DisputeManager`'s are automatically rejected. The semantic version is handled by the application layer (i.e., the graph node/indexer) and should thus live in the signature payload.

Note that there are multiple ways to accomplish this. However, the important part is that `chainId` and `address(this)` must be checked (either explicitly in code or implicitly as part of the domain separator; the latter is preferred) to prevent replay attacks.

3.2 DisputeManager - Subgraph versioning may lead to indexer partitioning Medium

Description

PR 548/gip-0024 introduces semantic API versioning, which might degrade the security of the system assuming indexer nodes lag behind, not upgrading to serve the latest versions. This can lead to specific versions only being served by a minority of nodes (or just a single one). The result of this query can be undisputable.

Introducing partitions to the set of indexer nodes serving a sub-graph degrades the system's security, assuming that a user querying the graph should not be able to control which specific node returns the result or how many nodes are actually serviced that subgraph version partition.

`https://gateway.thegraph.com/api/[api-key]/subgraphs/id/[identifier]?api-version=[semver]`

It should also be noted that users might be forced into a partition by a malicious Man-in-the-Middle changing the api-version in-flight of non-transport-secured HTTP requests. It is highly recommended to enforce that indexer queries are always transport-secured (i.e., TLS encrypted, authenticated, ...).

Recommendation

Ensure that enough indexer nodes are servicing a subgraph api version at any given point.

3.3 DisputeManager - gas optimization: Use the `ECDSA.recover(hash, v, r, s)` overload instead of `ECDSA.recover(hash, bytes sig=abi.encodePacked(r, s, v))` Minor

Description

The current implementation uses `ECDSA.recover(hash, bytes sig)` where `sig=abi.encodePacked(r, s, v)`. However, there is an `ECDSA.recover(hash, v, r, s)` overload that removes the necessity to `abi.encode` completely, saving some gas.

pr548/contracts/disputes/DisputeManager.sol:L739-L742

```
ECDSA.recover(
    messageHash,
    abi.encodePacked(_attestation.r, _attestation.s, _attestation.v)
);
```

`r`, `s`, and `v` are already in the proper format for the `ECDSA.recover(hash, v, r, s)` overload below

pr548/contracts/disputes/IDisputeManager.sol:L32-L42

```
// Attestation sent from indexer in response to a request
struct Attestation {
    bytes32 requestCID;
    bytes32 responseCID;
    bytes32 subgraphDeploymentID;
    bytes32 r;
    bytes32 s;
    uint8 v;
    bytes32 domainSeparator;
}
```

For reference, here's a version of the OZ `ECDSA` library in use:

contracts/utils/cryptography/ECDSA.sol:L177-L190

```
/**
 * @dev Overload of {ECDSA-recover} that receives the `v`,
 * `r` and `s` signature fields separately.
 */
function recover(
    bytes32 hash,
    uint8 v,
    bytes32 r,
    bytes32 s
) internal pure returns (address) {
    (address recovered, RecoverError error) = tryRecover(hash, v, r, s);
    _throwError(error);
    return recovered;
}
```

```
    /**
     * @dev Overload of {ECDSA-recover} that receives the `v`,
     * `r` and `s` signature fields separately.
     */
    function recover(
        bytes32 hash,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) internal pure returns (address) {
        (address recovered, RecoverError error) = tryRecover(hash, v, r, s);
        _throwError(error);
        return recovered;
    }
```

Recommendation

Use `ECDSA.recover(messageHash, _attestation.v, _attestation.r, _attestation.s)`.

3.4 DisputeManager - No guarantee that Disputer's funds are ever released Minor Out of Scope

Description

The arbitrator (`onlyArbitrator`) seems to be a semi-manual role in the system. Hence, disputes might not be handled at all, which can lead to the disputing parties' funds being locked forever.

Examples

pr548/contracts/disputes/DisputeManager.sol:L35-L37

```
* Arbitration:
* Disputes can only be accepted, rejected or drawn by the arbitrator role that can be delegated
* to a EOA or DAO.
```

pr548/contracts/disputes/DisputeManager.sol:L354-L377

```
/**
 * @dev Create a query dispute for the arbitrator to resolve.
 * This function is called by a fisherman that will need to `_deposit` at
 * least `minimumDeposit` GRT tokens.
 * @param _attestationData Attestation bytes submitted by the fisherman
 * @param _deposit Amount of tokens staked as deposit
 */
function createQueryDispute(bytes calldata _attestationData, uint256 _deposit)
    external
    override
    returns (bytes32)
{
    // Get funds from submitter
    _pullSubmitterDeposit(_deposit);

    // Create a dispute
    return
        _createQueryDisputeWithAttestation(
            msg.sender,
            _deposit,
            _parseAttestation(_attestationData),
            _attestationData
        );
}
```

pr548/contracts/disputes/DisputeManager.sol:L498-L508

```
function createIndexingDispute(address _allocationID, uint256 _deposit)
    external
    override
    returns (bytes32)
{
    // Get funds from submitter
    _pullSubmitterDeposit(_deposit);

    // Create a dispute
    return _createIndexingDisputeWithAllocation(msg.sender, _deposit, _allocationID);
}
```

Recommendation

In order to provide a better incentive and security guarantee to someone disputing a query/index it is suggested to timelock the dispute and allow the disputer to cancel it if the arbitrator did not handle the dispute in time.

3.5 DisputeManager - `createQueryDisputeConflict` assumes one deposit it right Out of Scope

Description

It is unclear how the arbitrator decides if at least one of the attestations submitted to `createQueryDisputeConflict` is correct. According to the comment, it is assumed that one is correct, but we could not find more information about how this is being ensured.

It is, therefore, recommended to document this process and outline what steps are taken by the arbitrator to ensure that the dispute is only resolved for disputes where at least one attestation is correct.

Examples

pr548/contracts/disputes/DisputeManager.sol:L390-L395

```
 */
function createQueryDisputeConflict(
    bytes calldata _attestationData1,
    bytes calldata _attestationData2
) external override returns (bytes32, bytes32) {
    address fisherman = msg.sender;
```

pr548/contracts/disputes/DisputeManager.sol:L383-L384

```
* For this type of dispute the submitter is not required to present a deposit
* as one of the attestation is considered to be right.
```

Appendix 1 - Disclosure

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.