
# DAO Signer Review

**August 22, 2025**

Prepared for Uniswap Foundation

Conducted by:

Kurt Willis (phaze)

Richie Humphrey (devtooligan)

## About the Uniswap Foundation DAO Signer Review

The Uniswap Foundation is a nonprofit organization established in 2022 to support the growth, sustainability, and decentralization of the Uniswap Protocol and its ecosystem. Its mission is to foster a more open and fair financial system by funding innovation, supporting developers, and advancing decentralized finance (DeFi) governance.

This repository contains a system for creating, managing, and verifying two-party agreements on top of the Ethereum Attestation Service (EAS). It provides a standardized and secure way for a primary entity (such as a DAO, protocol, or individual) to enter into on-chain agreements with various counterparties. The core of the system is the `AgreementAnchor` contract, which serves as an immutable, on-chain record of an agreement about some off-chain content, identified by a `bytes32` content hash.

## About Offbeat Security

Offbeat Security is a boutique security company providing unique security solutions for complex and novel crypto projects. Our mission is to elevate the blockchain security landscape through invention and collaboration.

## Summary & Scope

The src folder was reviewed at commit 08b90c9.

The following **4 files** were in scope:

- src/AgreementAnchor.sol
- src/AgreementAnchorFactory.sol
- src/AgreementResolver.sol
- src/interfaces/IAgreementAnchorFactory.sol

## Summary of Findings

| Identifier | Title | Severity | Fixed |
|---|---|---|---|
| L-01 | Attestation expiration time is not validated causing agreement validity ambiguity | Low | Fixed in PR#4 |
| L-02 | Parties can front-run agreement completion by updating attestation notes | Low | Fixed in PR#4 |
| L-03 | Schema UID is not validated allowing bypass of agreement resolver schema | Low | Fixed in PR#4 |
| L-04 | Revocable agreements cause legal confusion and should be prohibited | Low | Fixed in PR#4 |
| I-01 | RefUID field is not validated potentially causing confusion about attestation relationships | Info | Ack |
| I-02 | Attestation data is not validated for proper string decoding | Info | Fixed in PR#4 |

## Additional Recommendations

### Code Quality

See Appendix A for code quality and protocol design recommendations.

## Detailed Findings

# Low Findings

## [L-01] Attestation expiration time is not validated causing agreement validity ambiguity

### Description

The AgreementResolver contract does not validate the `expirationTime` field of EAS attestations during the attestation process. This creates a potential ambiguity in determining agreement validity, as expired attestations may still be stored and referenced by the AgreementAnchor contract.

The `_enforceAttestationRules()` function validates several attestation properties but omits checking the expiration:

```
function _enforceAttestationRules(Attestation calldata attestation)
    internal
    view
    returns (address attester, AgreementAnchor anchor)
{
    // ... other validation checks ...

    // Optionally enforce attestation expiration = 0, etc
}
```

When an attestation expires, the EAS system may treat it as invalid, but the AgreementAnchor contract continues to store the expired attestation's UID in `partyA_attestationUID` or `partyB_attestationUID`. This creates uncertainty about whether an agreement should be considered valid when one or both parties' attestations have expired.

The ambiguity arises in scenarios such as:

- Party A's attestation expires before Party B attests
- Party A's attestation expires after both parties have attested
- Different interpretation of expired attestations by external systems

### Recommendation

Consider adding validation to ensure attestations either have no expiration or have appropriate expiration handling:

```
function _enforceAttestationRules(Attestation calldata attestation)
    internal
    view
    returns (address attester, AgreementAnchor anchor)
{
```

```
    // ... existing validation ...

-   // Optionally enforce attestation expiration = 0, etc
+   // Require no expiration to avoid validity ambiguity
+   require(attestation.expirationTime == 0, "Attestations cannot have expiratio
}
```

Alternatively, if expiration functionality is desired, implement clear logic for handling expired attestations within the AgreementAnchor contract and document the expected behavior when attestations expire at different stages of the agreement process.

## [L-02] Parties can front-run agreement completion by updating attestation notes

### Description

The current implementation allows parties to update their attestation notes by making multiple attestations, with only the most recent attestation UID being stored in the `AgreementAnchor`. This creates a front-running vulnerability where a party can maliciously change their attestation terms right before the counterparty completes the agreement.

The vulnerability occurs because:

1. The `onAttest()` function allows parties to attest multiple times as long as both parties haven't completed their attestations
2. Each new attestation from the same party overwrites the previous `partyA_attestationUID` or `partyB_attestationUID`
3. The attestation data (notes) can contain different terms or conditions

Consider this attack scenario:

- Party A creates an initial attestation with note "we agree to Option 1"
- Party B reviews the terms and decides to enter the agreement
- Right before Party B's transaction is mined, Party A front-runs with a new attestation containing "we agree to Option 2"
- Party B's transaction completes, but they are now bound to Option 2 instead of the Option 1 they intended to agree to

The `AgreementAnchor__AgreementAlreadyAttested()` check only prevents attestations after both parties have attested, but allows unlimited updates before agreement completion:

```
function onAttest(address party, bytes32 uid) external onlyResolver {
    if (didEitherPartyRevoke) revert AgreementAnchor__AgreementRevoked();
    if (partyA_attestationUID != 0x0 && partyB_attestationUID != 0x0) {
        revert AgreementAnchor__AgreementAlreadyAttested();
```

```
    }
    // Party can update their attestation multiple times before both parties atte
    if (party == PARTY_A) partyA_attestationUID = uid;
    else if (party == PARTY_B) partyB_attestationUID = uid;
}
```

### Recommendation

Consider implementing one of the following approaches to prevent note manipulation:

#### Option 1: Single attestation per party

```
function onAttest(address party, bytes32 uid) external onlyResolver {
    if (didEitherPartyRevoke) revert AgreementAnchor__AgreementRevoked();
    if (partyA_attestationUID != 0x0 && partyB_attestationUID != 0x0) {
        revert AgreementAnchor__AgreementAlreadyAttested();
    }
+
+   if (party == PARTY_A) {
+       require(partyA_attestationUID == 0x0, "Party A has already attested");
+       partyA_attestationUID = uid;
+   } else if (party == PARTY_B) {
+       require(partyB_attestationUID == 0x0, "Party B has already attested");
+       partyB_attestationUID = uid;
+   }
-   if (party == PARTY_A) partyA_attestationUID = uid;
-   else if (party == PARTY_B) partyB_attestationUID = uid;
    else revert AgreementAnchor__NotAParty();
}
```

#### Option 2: Remove notes functionality entirely

Removing the ability to include notes in attestations eliminates this attack vector while maintaining the core agreement functionality.

## [L-03] Schema UID is not validated allowing bypass of agreement resolver schema

### Description

The AgreementResolver contract does not validate the `schema` field of EAS attestations, which allows attackers to register malicious schemas that still use the `AgreementResolver` but with modified properties. This enables bypassing intended schema constraints without bypassing the resolver itself.

The `_enforceAttestationRules()` function validates several attestation properties but omits checking the schema UID:

```
function _enforceAttestationRules(Attestation calldata attestation)
    internal
    view
```

```
    returns (address attester, AgreementAnchor anchor)
{
    // ... validation checks for attester, anchor, content hash ...

    // No validation of attestation.schema
}
```

This oversight allows attackers to register alternative schemas that:

1. **Enable unauthorized revocations**: Register a schema with `revocable = true` while still using the AgreementResolver, allowing revocations even if the intended design prohibits them
2. **Use different data formats**: Register a schema with arbitrary data structure (e.g., different encoding than the expected `bytes32` content hash), potentially causing parsing issues or unexpected behavior
3. **Create schema confusion**: Multiple valid schemas can target the same AgreementAnchor contracts, making it unclear which schema represents the canonical agreement format

Since the resolver validation logic will still execute for these alternative schemas, the attestations will be considered valid by the AgreementAnchor contract, but they may not conform to the intended schema specifications.

### Proof of Concept

```
// Attacker registers a malicious schema targeting the same resolver
bytes32 maliciousSchema = schemaRegistry.register(
    "uint256 fakeData", // Different data format than expected
    ISchemaResolver(address(agreementResolver)), // Still uses the same resolver
    true // Allow revocations even if not intended
);

// Create attestation that passes resolver validation but uses wrong schema
eas.attest(AttestationRequest({
    schema: maliciousSchema, // Uses schema with unintended properties
    recipient: address(legitimateAgreementAnchor),
    data: abi.encode(uint256(123)), // Wrong data format
    // ... other fields
}));

// The resolver validation will execute and may incorrectly decode the data
// or allow revocations when they shouldn't be permitted
```

### Recommendation

The `AgreementResolver` should compute and validate the expected schema UID. Consider implementing the following changes:

```
contract AgreementResolver is SchemaResolver {
+   bytes32 public immutable SCHEMA_UID;
+
    constructor(IEAS eas, address _signer) SchemaResolver(eas) {
+       // Compute the expected schema UID
+       string memory schemaDefinition = "bytes32 contentHash,string note";
+       SCHEMA_UID = keccak256(abi.encodePacked(
+           schemaDefinition,
+           ISchemaResolver(address(this)),
+           false // revocable = false
+       ));
+
+       // Check if schema exists, if not register it
+       ISchemaRegistry schemaRegistry = eas.getSchemaRegistry();
+       SchemaRecord memory existingSchema = schemaRegistry.getSchema(SCHEMA_UID)
+       if (existingSchema.uid != SCHEMA_UID) {
+           bytes32 registeredUID = schemaRegistry.register(schemaDefinition, ISc
+           require(registeredUID == SCHEMA_UID, "Schema UID mismatch");
+       }
+
        ANCHOR_FACTORY = new AgreementAnchorFactory(address(this), _signer);
    }

    function _enforceAttestationRules(Attestation calldata attestation)
        internal
        view
        returns (address attester, AgreementAnchor anchor)
    {
+       // Ensure attestation uses the correct schema
+       require(attestation.schema == SCHEMA_UID, "Invalid schema UID");

        // ... existing validation ...
    }
}
```

This approach ensures that only attestations using the specific schema with the correct resolver and revocable settings can interact with AgreementAnchor contracts.

## [L-04] Revocable agreements cause legal confusion and should be prohibited

### Description

The current system allows for the creation of revocable agreement schemas, which introduces legal confusion and ambiguity about agreement validity. While technically possible through EAS, revocable agreements create problematic scenarios where the on-chain status may not align with legal reality.

The fundamental issue is that revocable attestations introduce "quasi-legal" concepts that create tension between on-chain and off-chain legal interpretations. This leads to several problematic scenarios:

1. **Legal vs. technical revocations**: A party might revoke an attestation on-chain without legal justification, creating confusion about whether the agreement is actually terminated
2. **Malicious revocations**: A party could revoke their attestation strategically (e.g., front-running the counterparty's attestation) to gain an unfair advantage
3. **Multiple sources of truth**: Both the EAS revocation status and the AgreementAnchor's `didEitherPartyRevoke` flag could provide conflicting information about agreement validity
4. **Legal enforceability questions**: Courts would need to determine whether on-chain revocations have legal effect, especially when done without proper legal basis

The question whether on-chain revocations should have binding legal effects remains. Disputes would ultimately need to be resolved in real life. This creates an uncomfortable situation where the on-chain state may not reflect the true legal status of an agreement.

The preferred approach, is to eliminate revocations entirely to provide clear legal certainty and avoid these complex edge cases.

#### Recommendation

Prohibit revocable agreements by ensuring all schemas used for agreements are non-revocable. The AgreementResolver should enforce this constraint:

```
contract AgreementResolver is SchemaResolver {
+   bytes32 public immutable SCHEMA_UID;
+
    constructor(IEAS eas, address _signer) SchemaResolver(eas) {
+       // Define and register a non-revocable schema
+       string memory schemaDefinition = "bytes32 contentHash,string note";
+       SCHEMA_UID = keccak256(abi.encodePacked(
+           schemaDefinition,
+           ISchemaResolver(address(this)),
+           false // revocable = false - legally required
+       ));
+
+       ISchemaRegistry schemaRegistry = eas.getSchemaRegistry();
+       SchemaRecord memory existingSchema = schemaRegistry.getSchema(SCHEMA_UID)
+       if (existingSchema.uid != SCHEMA_UID) {
+           bytes32 registeredUID = schemaRegistry.register(schemaDefinition, ISc
+           require(registeredUID == SCHEMA_UID, "Schema UID mismatch");
+       }
+
        ANCHOR_FACTORY = new AgreementAnchorFactory(address(this), _signer);
    }

    function _enforceAttestationRules(Attestation calldata attestation)
        internal
        view
        returns (address attester, AgreementAnchor anchor)
    {
```

```
+        // Ensure only non-revocable agreements are allowed
+        require(attestation.schema == SCHEMA_UID, "Must use non-revocable schema"

         // ... existing validation ...
     }
}
```

Additionally, consider removing the revocation logic from the AgreementAnchor contract entirely:

```
- function onRevoke(address party, bytes32 uid) external onlyResolver {
-     didEitherPartyRevoke = true;
-     emit PartyRevoked(party, uid);
- }
```

## Informational Findings

### [I-01] RefUID field is not validated potentially causing confusion about attestation relationships

#### Description

The AgreementResolver contract does not validate the `refUID` field of EAS attestations, which could lead to confusion about the relationships between attestations and agreements. The `refUID` field is designed to reference previous attestations, but in the context of DUNI agreements, its use case and validation requirements are unclear.

The `_enforceAttestationRules()` function validates several attestation properties but does not check the `refUID` field:

```
function _enforceAttestationRules(Attestation calldata attestation)
    internal
    view
    returns (address attester, AgreementAnchor anchor)
{
    // ... validation checks for attester, anchor, content hash ...

    // No validation of attestation.refUID
}
```

This omission creates several potential issues:

1. **Ambiguous references**: Attestations could reference unrelated previous attestations, creating confusion about the logical flow of agreements
2. **Invalid relationships**: A `refUID` could point to attestations from completely different agreement contexts or even non-agreement attestations

3. **Misleading connections**: External systems interpreting these references might incorrectly assume relationships between unrelated agreements

While `refUID` could potentially be useful for referencing previous agreement versions (especially if revocations are removed in favor of new agreement attestations), the current implementation provides no guarantees about the validity or relevance of these references.

If `refUID` references are allowed to point to previous AgreementAnchors, they should be validated to ensure they originated from the same AgreementAnchorFactory to maintain logical consistency.

**Recommendation**

Consider adding validation to ensure `refUID` is either empty or points to valid related attestations:

**Option 1: Require empty refUID**

```
function _enforceAttestationRules(Attestation calldata attestation)
    internal
    view
    returns (address attester, AgreementAnchor anchor)
{
    // ... existing validation ...

+   // Require no reference to avoid ambiguous relationships
+   require(attestation.refUID == bytes32(0), "RefUID must be empty for agreement
}
```

**Option 2: Validate refUID points to same factory's agreements (if allowing references)**

```
function _enforceAttestationRules(Attestation calldata attestation)
    internal
    view
    returns (address attester, AgreementAnchor anchor)
{
    // ... existing validation ...

+   // If refUID is provided, ensure it references an attestation from the same
+   if (attestation.refUID != bytes32(0)) {
+       Attestation memory refAttestation = _eas.getAttestation(attestation.refUI
+       require(
+           ANCHOR_FACTORY.isFactoryDeployed(refAttestation.recipient),
+           "RefUID must reference attestation from same factory"
+       );
+   }
}
```

## [I-02] Attestation data is not validated for proper string decoding

### Description

The AgreementResolver contract does not validate that the attestation data can be properly decoded according to the expected schema format. The schema expects `(bytes32 contentHash, string note)` but the current implementation only validates the content hash portion, potentially allowing malformed or invalid note data.

In `_enforceAttestationRules()`, only the content hash is validated:

```
require(
    abi.decode(attestation.data, (bytes32)) == anchor.CONTENT_HASH(),
    "Attestation data does not match the anchor"
);
```

This validation only decodes the first `bytes32` from the data, but does not verify that the remaining data can be properly decoded as a string. Invalid or malformed note data could cause issues for off-chain systems attempting to parse the attestation data.

Additionally, the validated content hash and note could be emitted in an event to improve off-chain tracking and verification of agreement details.

### Recommendation

Validate that the attestation data can be properly decoded and emit the parsed values:

```
function _enforceAttestationRules(Attestation calldata attestation)
    internal
    view
    returns (address attester, AgreementAnchor anchor)
{
    attester = attestation.attester;
    anchor = AgreementAnchor(attestation.recipient);

    // The anchor must have been deployed by this factory
    require(ANCHOR_FACTORY.isFactoryDeployed(address(anchor)), "Not a factory-dep

    // The attester must be one of the two parties defined in the anchor
    require(
        attester == anchor.PARTY_A() || attester == anchor.PARTY_B(), "Not a part
    );

-   // Attestation content hash must match anchor content hash
-   require(
-       abi.decode(attestation.data, (bytes32)) == anchor.CONTENT_HASH(),
-       "Attestation data does not match the anchor"
-   );
+   // Validate and decode attestation data
+   (bytes32 contentHash, string memory note) = abi.decode(attestation.data, (byt
+   require(contentHash == anchor.CONTENT_HASH(), "Attestation data does not mat
+
+   // Emit event with parsed data for off-chain tracking
```

```
+    emit AttestationValidated(attester, address(anchor), contentHash, note);
}
```

# Appendix A: Code Quality Recommendations

## Design and Code Quality Improvements

1. **Merge AgreementResolver and AgreementAnchorFactory**

   The two contracts are tightly coupled and managing them separately adds unnecessary complexity. Consider making AgreementResolver inherit from AgreementAnchorFactory:

   ```
   contract AgreementResolver is SchemaResolver, IAgreementAnchorFactory {
       // Combines both resolver and factory functionality
       // Eliminates need for separate ANCHOR_FACTORY reference
   }
   ```

2. **Expose Schema Information Functions**

   Add public functions to improve transparency and integration:

   ```
   function getSchemaDefinition() external pure returns (string memory) {
       return "bytes32 contentHash,string note";
   }

   function getSchemaUID() external view returns (bytes32) {
       return SCHEMA_UID;
   }

   function getSchemaRecord() external view returns (SchemaRecord memory) {
       return eas.getSchemaRegistry().getSchema(SCHEMA_UID);
   }
   ```

3. **Use CREATE2 for Deterministic Agreement Addresses**

   Implement CREATE2 deployment (using clones if desired) to enable canonical addresses for agreements and prevent duplicate agreements.

4. **Add Clear Agreement Validity Function**

   Provide a simple function to check if an agreement is valid:

   ```
   function isAgreementValid(address agreementAnchor) external view returns (boo
       AgreementAnchor anchor = AgreementAnchor(agreementAnchor);
       return anchor.partyA_attestationUID() != bytes32(0) &&
   ```

```
            anchor.partyB_attestationUID() != bytes32(0) &&
            !anchor.didEitherPartyRevoke();
}
```

5. **Use Consistent Error Handling**

   Replace mixed error styles with consistent custom errors:

```
- require(ANCHOR_FACTORY.isFactoryDeployed(address(anchor)), "Not a factory-c
+ if (!ANCHOR_FACTORY.isFactoryDeployed(address(anchor))) {
+     revert AgreementResolver__NotFactoryDeployed();
+ }

- require(attester == anchor.PARTY_A() || attester == anchor.PARTY_B(), "Not
+ if (attester != anchor.PARTY_A() && attester != anchor.PARTY_B()) {
+     revert AgreementResolver__NotAPartyToAgreement();
+ }
```