METATRUST

Pre Report for

bas-contract

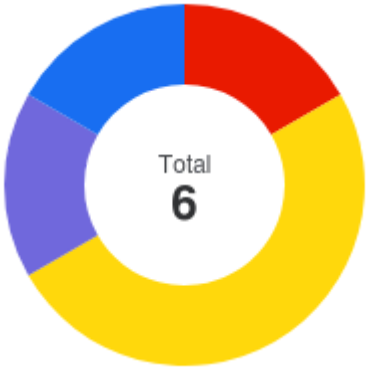
April 29, 2024

Executive Summary

Overview	
Project Name	bas-contract
Codebase URL	/
Scan Engine	Security Analyzer
Scan Time	2024/04/29 23:20:18
Commit Id	-

Total	
Critical Issues	1
High risk Issues	0
Medium risk Issues	3
Low risk Issues	1
Informational Issues	1

Critical Issues	The issue can cause large economic losses, large-scale data disorder, loss of control of authority management, failure of key functions, or indirectly affect the correct operation of other smart contracts interacting with it.
High Risk Issues	The issue puts a large number of users' sensitive information at risk or is reasonably likely to lead to catastrophic impacts on clients' reputations or serious financial implications for clients and users.
Medium Risk Issues	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact.
Low Risk Issues	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low-impact in view of the client's business circumstances.
Informational Issue	The issue does not pose an immediate risk but is relevant to security best practices or Defence in Depth.



Critical Issues	17%	1
High risk Issues	0%	0
Medium risk Issues	50%	3
Low risk Issues	17%	1
Informational Issues	17%	1

Summary of Findings

MetaScan security assessment was performed on **April 29, 2024 23:20:18** on project **bas-contract** with the repository on branch **default branch**. The assessment was carried out by scanning the project's codebase using the scan engine **Security Analyzer**. There are in total **6** vulnerabilities / security risks discovered during the scanning session, among which **1** critical vulnerabilities, **3** medium risk vulnerabilities, **1** low risk vulnerabilities, **1** informational issues.

ID	Description	Severity
MSA-001	Incorrect Verification Logic in `_verifyAttester` Allows Privilege Escalation	Critical
MSA-002	Incorrect Handling of Reverse Order Slicing in `_sliceUIDs` Function	Medium risk
MSA-003	Griefing Vulnerability in Timestamp Setting Logic	Medium risk
MSA-004	Lack of Validation for Self-Issued Attestations	Medium risk
MSA-005	MWE-105: Missing Zero Address Check	Low risk
MSA-006	MWE-110: Missing Event Setter	Informational

Findings

Critical (1)

1. Incorrect Verification Logic in `_verifyAttester` Allows Privilege Escalation

Critical | Security Analyzer

The `_verifyAttester` function within the smart contract is intended to restrict certain actions, specifically the delegation of attestations, to the contract owner only. This function checks if the `attester` parameter provided matches the owner's address. However, the critical flaw arises because this function does not verify that the caller of the transaction (`msg.sender`) is the contract owner. Instead, it relies on the `attester` parameter that is passed through `DelegatedProxyAttestationRequest`, which is user-controlled. This misalignment in the logic allows an attacker to spoof the identity by setting the `attester` field to the owner's address, even if they are not the owner, thus bypassing the intended security check.

The exploit occurs as follows:

1. An attacker crafts a `DelegatedProxyAttestationRequest` with the `attester` field set to the owner's address.
2. The attacker calls `attestByDelegation`, passing the crafted request.
3. The `_verifyAttester` checks if the `attester` matches the owner's address, which it does due to the attacker's manipulation, and not if `msg.sender` is the owner.
4. As a result, the attacker can perform actions meant only for the owner, exploiting the function without needing ownership credentials.

File(s) Affected

bas-contract-main/contracts/eip712/proxy/examples/PermissionedEIP712Proxy.sol #74-78

```
74      function _verifyAttester(address attester) private view {
75          if (attester != owner()) {
76              revert AccessDenied();
77          }
78      }
```

bas-contract-main/contracts/eip712/proxy/examples/PermissionedEIP712Proxy.sol #29-36

```
29      function attestByDelegation(
30          DelegatedProxyAttestationRequest calldata delegatedRequest
31      ) public payable override returns (bytes32) {
32          // Ensure that only the owner is allowed to delegate attestations.
33          _verifyAttester(delegatedRequest.attester);
34
35          return super.attestByDelegation(delegatedRequest);
36      }
```

Recommendation

To mitigate this vulnerability, the smart contract should be updated to verify that the caller of the function (`msg.sender`) is the owner, rather than relying on the potentially manipulated `attester` field. This ensures that only the true owner can invoke actions requiring owner privileges. ```solidity function _verifyAttester() private view { // Verify that the caller of the function is the contract's owner if (msg.sender != owner()) { revert AccessDenied("Only the owner can perform this action."); } }

function attestByDelegation(DelegatedProxyAttestationRequest calldata delegatedRequest) public payable override returns (bytes32) { // Ensure that only the owner is allowed to delegate attestations. _verifyAttester(); // Now checks `msg.sender` instead of using `delegatedRequest.attester`

```
    return super.attestByDelegation(delegatedRequest);
}
```

```
```

Medium risk (3)

1. Incorrect Handling of Reverse Order Slicing in `_sliceUIDs` Function

Medium risk | Security Analyzer

The `_sliceUIDs` function in the provided Solidity code is responsible for returning a slice of an array of UIDs based on the given `start`, `length`, and `reverseOrder` parameters. However, upon careful analysis, it has been discovered that the function fails to correctly handle scenarios where the requested slice range exceeds the array's bounds when `reverseOrder` is set to `true`.

The vulnerability lies in the following lines of code:

```
unchecked {
    uint256 len = length;
    if (attestationsLength < start + length) {
        len = attestationsLength - start;
    }

    bytes32[] memory res = new bytes32[](len);

    for (uint256 i = 0; i < len; ++i) {
        res[i] = uids[reverseOrder ? attestationsLength - (start + i + 1) : start + i];
    }

    return res;
}
```

When `reverseOrder` is `true`, and the requested `start` and `length` parameters exceed the `attestationsLength`, the function adjusts the `len` variable to ensure the slice fits within the array's bounds. However, it fails to update the `start` index accordingly, leading to incorrect element access during the reverse slicing process.

For example, consider a scenario where `uids.length` is 10, `start` is 8, `length` is 5, and `reverseOrder` is `true`. The function correctly adjusts `len` to 2 to prevent out-of-bounds access. However, when constructing the slice in reverse order, it starts accessing elements from `uids[1]` and `uids[0]` instead of the expected `uids[9]` and `uids[8]`.

This vulnerability can lead to incorrect data retrieval and potential misrepresentation of information when using the `_sliceUIDs` function with `reverseOrder` set to `true` and the requested range exceeding the array's bounds.

File(s) Affected

bas-contract-main/contracts/Indexer.sol #222-251

```
222     function _sliceUIDs(
223         bytes32[] memory uids,
224         uint256 start,
225         uint256 length,
226         bool reverseOrder
227     ) private pure returns (bytes32[] memory) {
228         uint256 attestationsLength = uids.length;
229         if (attestationsLength == 0) {
230             return new bytes32[](0);
231         }
232 
233         if (start >= attestationsLength) {
234             revert InvalidOffset();
235         }
236 
237         unchecked {
238             uint256 len = length;
239             if (attestationsLength < start + length) {
240                 len = attestationsLength - start;
241             }
242 
243             bytes32[] memory res = new bytes32[](len);
244 
245             for (uint256 i = 0; i < len; ++i) {
246                 res[i] = uids[reverseOrder ? attestationsLength - (start + i + 1) : start + i];
247             }
248 
249             return res;
250         }
251     }
```

Recommendation

To address this vulnerability and ensure correct handling of reverse order slicing, it is recommended to modify the `_sliceUIDs` function as follows:

```
unchecked {
    uint256 len = length;
    if (attestationsLength < start + length) {
        len = attestationsLength > start ? attestationsLength - start : 0;
    }

    bytes32[] memory res = new bytes32[](len);

    if (reverseOrder && len > 0) {
        uint256 endIndex = start + len;
        for (uint256 i = 0; i < len; ++i) {
            res[i] = uids[attestationsLength - (endIndex - i)];
        }
    } else {
        for (uint256 i = 0; i < len; ++i) {
            res[i] = uids[start + i];
        }
    }

    return res;
}
```

The modified code introduces the following changes:

1. When adjusting `len` due to the requested range exceeding the array's bounds, it ensures that `len` is set to 0 if `start` is already greater than or equal to `attestationsLength`.
2. It separates the reverse order slicing logic into a separate conditional block.
3. It calculates `endIndex` as `start + len` to determine the correct end index for the slice.
4. It uses `endIndex` in the reverse order slicing loop to ensure the correct elements are accessed.

2. Griefing Vulnerability in Timestamp Setting Logic

Medium risk Security Analyzer

The provided Solidity code contains a vulnerability in the logic for setting timestamps on specific pieces of data. The `_timestamp` function is designed to associate a timestamp with a given `bytes32` data, but it can only do so once for each unique data due to the following check:

```
if (_timestamps[data] != 0) {
    revert AlreadyTimestamped();
}
```

While this restriction is intended to prevent overwriting existing timestamps, it can be exploited by an attacker to permanently prevent a legitimate user from setting a timestamp on a specific piece of data. The attacker can achieve this by frontrunning the legitimate user's transaction and calling the `_timestamp` function with the same data before the user's transaction is confirmed.

Here's a step-by-step breakdown of how the vulnerability can be exploited:

1. The attacker monitors the blockchain for transactions that are about to call the `_timestamp` function (or any public/external function that leads to `_timestamp` being called) with a specific piece of data.
2. Before the legitimate user's transaction is confirmed, the attacker creates a transaction with the same data and a valid timestamp, and sends it with a higher gas price to ensure it is processed first.
3. Once the attacker's transaction is confirmed, the `_timestamps` mapping will have a non-zero value for the targeted data, preventing any future attempts to set a timestamp for that data.
4. When the legitimate user's transaction is processed, it will revert due to the `AlreadyTimestamped()` check, effectively griefing the user and preventing them from setting the intended timestamp.

File(s) Affected

bas-contract-main/contracts/EAS.sol #729-737

```
729     function _timestamp(bytes32 data, uint64 time) private {
730         if (_timestamps[data] != 0) {
731             revert AlreadyTimestamped();
732         }
733
734         _timestamps[data] = time;
735
736         emit Timestamped(data, time);
737     }
```

Recommendation

To mitigate this vulnerability, it is recommended to modify the `_timestamp` function to allow for updating existing timestamps under certain conditions. This can be achieved by replacing the `AlreadyTimestamped()` revert with a conditional update logic. Here's an updated version of the `_timestamp` function:

```
function _timestamp(bytes32 data, uint64 time) private {
    uint64 currentTimestamp = _timestamps[data];
    if (currentTimestamp != 0) {
        // Only allow updating the timestamp if the new timestamp is greater than the current one
        if (time <= currentTimestamp) {
            revert InvalidTimestamp();
        }
    }

    _timestamps[data] = time;
    emit Timestamped(data, time);
}
```

3. Lack of Validation for Self-Issued Attestations

Medium risk Security Analyzer

The `_indexAttestation` function in the provided code is vulnerable to exploitation due to the absence of a validation check to ensure that the `attester` and `recipient` of an attestation are not the same entity. This vulnerability allows an attacker to issue attestations to themselves, potentially manipulating metrics, rewards, or governance proceedings that rely on the count of sent and received attestations.

The core of the issue lies in the indexing logic:

```
_schemaAttestations[schema].push(attestationUID);
_receivedAttestations[recipient][schema].push(attestationUID);
_sentAttestations[attester][schema].push(attestationUID);
_schemaAttesterRecipientAttestations[schema][attester][recipient].push(attestationUID);
```

Here, the attestation is indexed based on the `schema`, `recipient`, and `attester`. However, there is no check to prevent a scenario where `attester` == `recipient`. This means an attacker can craft an attestation where they are both the sender and the receiver.

The impact of this vulnerability depends on how the indexed attestations are used within the broader context of the smart contract system. If the number of sent or received attestations influences governance decisions, rewards distribution, or other critical functionalities, an attacker could exploit this vulnerability to gain an undue advantage.

File(s) Affected

bas-contract-main/contracts/Indexer.sol #188-214

```
188     function _indexAttestation(bytes32 attestationUID) private {
189         // Skip already indexed attestations.
190         if (_indexedAttestations[attestationUID]) {
191             return;
192         }
193 
194         // Check if the attestation exists.
195         Attestation memory attestation = _eas.getAttestation(attestationUID);
196 
197         bytes32 uid = attestation.uid;
198         if (uid == EMPTY_UID) {
199             revert InvalidAttestation();
200         }
201 
202         // Index the attestation.
203         address attester = attestation.attester;
204         address recipient = attestation.recipient;
205         bytes32 schema = attestation.schema;
206 
207         _indexedAttestations[attestationUID] = true;
208         _schemaAttestations[schema].push(attestationUID);
209         _receivedAttestations[recipient][schema].push(attestationUID);
210         _sentAttestations[attester][schema].push(attestationUID);
211         _schemaAttesterRecipientAttestations[schema][attester][recipient].push(attestationUID);
212 
213         emit Indexed({ uid: uid });
214     }
```

Recommendation

To mitigate this vulnerability, it is recommended to add a validation check in the `_indexAttestation` function to ensure that the `attester` and `recipient` are not the same. This can be achieved by adding the following code before the indexing logic:

```
if (attester == recipient) {
    revert InvalidAttestationSelfSent();
}
```

Here's the updated `_indexAttestation` function with the recommended fix:

```
function _indexAttestation(bytes32 attestationUID) private {
    // Skip already indexed attestations.
    if (_indexedAttestations[attestationUID]) {
        return;
    }

    // Check if the attestation exists.
    Attestation memory attestation = _eas.getAttestation(attestationUID);

    bytes32 uid = attestation.uid;
    if (uid == EMPTY_UID) {
        revert InvalidAttestation();
    }

    address attester = attestation.attester;
    address recipient = attestation.recipient;

    // Check if attester and recipient are the same.
    if (attester == recipient) {
        revert InvalidAttestationSelfSent();
    }

    // Index the attestation.
    bytes32 schema = attestation.schema;

    _indexedAttestations[attestationUID] = true;
    _schemaAttestations[schema].push(attestationUID);
    _receivedAttestations[recipient][schema].push(attestationUID);
    _sentAttestations[attester][schema].push(attestationUID);
    _schemaAttesterRecipientAttestations[schema][attester][recipient].push(attestationUID);

    emit Indexed({ uid: uid });
}
```

Low risk (1)

1. MWE-105: Missing Zero Address Check

Low risk Security Analyzer

This Function is lack of zero address check in important operation, which may cause some unexpected result.

File(s) Affected

bas-contract-main/contracts/resolver/examples/RecipientResolver.sol #14-16

```
14      constructor(IEAS eas, address targetRecipient) SchemaResolver(eas) {
15          _targetRecipient = targetRecipient;
16      }
```

Recommendation

Add check of zero address in important operation.

Informational (1)

1. MWE-110: Missing Event Setter

Informational Security Analyzer

Setter-functions must emit events

File(s) Affected

bas-contract-main/contracts/EAS.sol #97-102

```
97      function attest(AttestationRequest calldata request) external payable returns (bytes32) {
98          AttestationRequestData[] memory data = new AttestationRequestData[](1);
99          data[0] = request.data;
100
101         return _attest(request.schema, data, msg.sender, msg.value, true).uids[0];
102     }
```

bas-contract-main/contracts/EAS.sol #105-114

```
105     function attestByDelegation(
106         DelegatedAttestationRequest calldata delegatedRequest
107     ) external payable returns (bytes32) {
108         _verifyAttest(delegatedRequest);
109
110         AttestationRequestData[] memory data = new AttestationRequestData[](1);
111         data[0] = delegatedRequest.data;
112
113         return _attest(delegatedRequest.schema, data, delegatedRequest.attester, msg.value, true).uids
114     }
```

bas-contract-main/contracts/EAS.sol #117-167

```
117     function multiAttest(MultiAttestationRequest[] calldata multiRequests) external payable returns (by
118         // Since a multi-attest call is going to make multiple attestations for multiple schemas, we'd
119         // all the returned UIDs into a single list.
120         uint256 length = multiRequests.length;
121         bytes32[][] memory totalUIDs = new bytes32[][](length);
122         uint256 totalUIDCount = 0;
123 
124         // We are keeping track of the total available ETH amount that can be sent to resolvers and wi
125         // from it to verify that there isn't any attempt to send too much ETH to resolvers. Please not
126         // some ETH was stuck in the contract by accident (which shouldn't happen in normal conditions,
127         // possible to send too much ETH anyway.
128         uint256 availableValue = msg.value;
129 
130         for (uint256 i = 0; i < length; i = uncheckedInc(i)) {
131             // The last batch is handled slightly differently: if the total available ETH wasn't spent
132             // is a remainder - it will be refunded back to the attester (something that we can only ve
133             // last and final batch).
134             bool last;
135             unchecked {
136                 last = i == length - 1;
137             }
138 
139             // Process the current batch of attestations.
140             MultiAttestationRequest calldata multiRequest = multiRequests[i];
141 
142             // Ensure that data isn't empty.
143             if (multiRequest.data.length == 0) {
144                 revert InvalidLength();
145             }
146 
147             AttestationsResult memory res = _attest(
148                 multiRequest.schema,
149                 multiRequest.data,
150                 msg.sender,
151                 availableValue,
152                 last
153             );
154 
155             // Ensure to deduct the ETH that was forwarded to the resolver during the processing of thi
156             availableValue -= res.usedValue;
157 
158             // Collect UIDs (and merge them later).
159             totalUIDs[i] = res.uids;
160             unchecked {
161                 totalUIDCount += res.uids.length;
162             }
163         }
164 
165         // Merge all the collected UIDs and return them as a flatten array.
166         return _mergeUIDs(totalUIDs, totalUIDCount);
167     }
```

bas-contract-main/contracts/EAS.sol #170-237

```
    function multiAttestByDelegation(
        MultiDelegatedAttestationRequest[] calldata multiDelegatedRequests
    ) external payable returns (bytes32[] memory) {
        // Since a multi-attest call is going to make multiple attestations for multiple schemas, we'd
        // all the returned UIDs into a single list.
        uint256 length = multiDelegatedRequests.length;
        bytes32[][] memory totalUIDs = new bytes32[][](length);
        uint256 totalUIDCount = 0;

        // We are keeping track of the total available ETH amount that can be sent to resolvers and wi
        // from it to verify that there isn't any attempt to send too much ETH to resolvers. Please not
        // some ETH was stuck in the contract by accident (which shouldn't happen in normal conditions,
        // possible to send too much ETH anyway.
        uint256 availableValue = msg.value;

        for (uint256 i = 0; i < length; i = uncheckedInc(i)) {
            // The last batch is handled slightly differently: if the total available ETH wasn't spent
            // is a remainder - it will be refunded back to the attester (something that we can only ve
            // last and final batch).
            bool last;
            unchecked {
                last = i == length - 1;
            }

            MultiDelegatedAttestationRequest calldata multiDelegatedRequest = multiDelegatedRequests[i]
            AttestationRequestData[] calldata data = multiDelegatedRequest.data;

            // Ensure that no inputs are missing.
            uint256 dataLength = data.length;
            if (dataLength == 0 || dataLength != multiDelegatedRequest.signatures.length) {
                revert InvalidLength();
            }

            // Verify signatures. Please note that the signatures are assumed to be signed with increas
            for (uint256 j = 0; j < dataLength; j = uncheckedInc(j)) {
                _verifyAttest(
                    DelegatedAttestationRequest({
                        schema: multiDelegatedRequest.schema,
                        data: data[j],
                        signature: multiDelegatedRequest.signatures[j],
                        attester: multiDelegatedRequest.attester,
                        deadline: multiDelegatedRequest.deadline
                    })
                );
            }

            // Process the current batch of attestations.
            AttestationsResult memory res = _attest(
                multiDelegatedRequest.schema,
                data,
                multiDelegatedRequest.attester,
                availableValue,
                last
            );

            // Ensure to deduct the ETH that was forwarded to the resolver during the processing of th
            availableValue -= res.usedValue;
```

```
227
228             // Collect UIDs (and merge them later).
229             totalUIDs[i] = res.uids;
230             unchecked {
231                 totalUIDCount += res.uids.length;
232             }
233         }
234
235         // Merge all the collected UIDs and return them as a flatten array.
236         return _mergeUIDs(totalUIDs, totalUIDCount);
237     }
```

bas-contract-main/contracts/EAS.sol #240-245

```
240     function revoke(RevocationRequest calldata request) external payable {
241         RevocationRequestData[] memory data = new RevocationRequestData[](1);
242         data[0] = request.data;
243
244         _revoke(request.schema, data, msg.sender, msg.value, true);
245     }
```

bas-contract-main/contracts/EAS.sol #248-255

```
248     function revokeByDelegation(DelegatedRevocationRequest calldata delegatedRequest) external payable
249         _verifyRevoke(delegatedRequest);
250
251         RevocationRequestData[] memory data = new RevocationRequestData[](1);
252         data[0] = delegatedRequest.data;
253
254         _revoke(delegatedRequest.schema, data, delegatedRequest.revoker, msg.value, true);
255     }
```

bas-contract-main/contracts/EAS.sol #258-280

```
258     function multiRevoke(MultiRevocationRequest[] calldata multiRequests) external payable {
259         // We are keeping track of the total available ETH amount that can be sent to resolvers and wi
260         // from it to verify that there isn't any attempt to send too much ETH to resolvers. Please not
261         // some ETH was stuck in the contract by accident (which shouldn't happen in normal conditions,
262         // possible to send too much ETH anyway.
263         uint256 availableValue = msg.value;
264
265         uint256 length = multiRequests.length;
266         for (uint256 i = 0; i < length; i = uncheckedInc(i)) {
267             // The last batch is handled slightly differently: if the total available ETH wasn't spent
268             // is a remainder - it will be refunded back to the attester (something that we can only ve
269             // last and final batch).
270             bool last;
271             unchecked {
272                 last = i == length - 1;
273             }
274
275             MultiRevocationRequest calldata multiRequest = multiRequests[i];
276
277             // Ensure to deduct the ETH that was forwarded to the resolver during the processing of thi
278             availableValue -= _revoke(multiRequest.schema, multiRequest.data, msg.sender, availableValu
279         }
280     }
```

bas-contract-main/contracts/EAS.sol #283-333

```
283     function multiRevokeByDelegation(
284         MultiDelegatedRevocationRequest[] calldata multiDelegatedRequests
285     ) external payable {
286         // We are keeping track of the total available ETH amount that can be sent to resolvers and wi
287         // from it to verify that there isn't any attempt to send too much ETH to resolvers. Please not
288         // some ETH was stuck in the contract by accident (which shouldn't happen in normal conditions,
289         // possible to send too much ETH anyway.
290         uint256 availableValue = msg.value;
291 
292         uint256 length = multiDelegatedRequests.length;
293         for (uint256 i = 0; i < length; i = uncheckedInc(i)) {
294             // The last batch is handled slightly differently: if the total available ETH wasn't spent
295             // is a remainder - it will be refunded back to the attester (something that we can only ve
296             // last and final batch).
297             bool last;
298             unchecked {
299                 last = i == length - 1;
300             }
301 
302             MultiDelegatedRevocationRequest memory multiDelegatedRequest = multiDelegatedRequests[i];
303             RevocationRequestData[] memory data = multiDelegatedRequest.data;
304 
305             // Ensure that no inputs are missing.
306             uint256 dataLength = data.length;
307             if (dataLength == 0 || dataLength != multiDelegatedRequest.signatures.length) {
308                 revert InvalidLength();
309             }
310 
311             // Verify signatures. Please note that the signatures are assumed to be signed with increas
312             for (uint256 j = 0; j < dataLength; j = uncheckedInc(j)) {
313                 _verifyRevoke(
314                     DelegatedRevocationRequest({
315                         schema: multiDelegatedRequest.schema,
316                         data: data[j],
317                         signature: multiDelegatedRequest.signatures[j],
318                         revoker: multiDelegatedRequest.revoker,
319                         deadline: multiDelegatedRequest.deadline
320                     })
321                 );
322             }
323 
324             // Ensure to deduct the ETH that was forwarded to the resolver during the processing of thi
325             availableValue -= _revoke(
326                 multiDelegatedRequest.schema,
327                 data,
328                 multiDelegatedRequest.revoker,
329                 availableValue,
330                 last
331             );
332         }
333     }
```

bas-contract-main/contracts/EAS.sol #345-351

```
345     function revokeOffchain(bytes32 data) external returns (uint64) {
346         uint64 time = _time();
347 
348         _revokeOffchain(msg.sender, data, time);
349 
350         return time;
351     }
```

bas-contract-main/contracts/EAS.sol #354-363

```
354     function multiRevokeOffchain(bytes32[] calldata data) external returns (uint64) {
355         uint64 time = _time();
356 
357         uint256 length = data.length;
358         for (uint256 i = 0; i < length; i = uncheckedInc(i)) {
359             _revokeOffchain(msg.sender, data[i], time);
360         }
361 
362         return time;
363     }
```

bas-contract-main/contracts/EAS.sol #561-613

```
561     function _resolveAttestation(
562         SchemaRecord memory schemaRecord,
563         Attestation memory attestation,
564         uint256 value,
565         bool isRevocation,
566         uint256 availableValue,
567         bool last
568     ) private returns (uint256) {
569         ISchemaResolver resolver = schemaRecord.resolver;
570         if (address(resolver) == address(0)) {
571             // Ensure that we don't accept payments if there is no resolver.
572             if (value != 0) {
573                 revert NotPayable();
574             }
575 
576             if (last) {
577                 _refund(availableValue);
578             }
579 
580             return 0;
581         }
582 
583         // Ensure that we don't accept payments which can't be forwarded to the resolver.
584         if (value != 0) {
585             if (!resolver.isPayable()) {
586                 revert NotPayable();
587             }
588 
589             // Ensure that the attester/revoker doesn't try to spend more than available.
590             if (value > availableValue) {
591                 revert InsufficientValue();
592             }
593 
594             // Ensure to deduct the sent value explicitly.
595             unchecked {
596                 availableValue -= value;
597             }
598         }
599 
600         if (isRevocation) {
601             if (!resolver.revoke{ value: value }(attestation)) {
602                 revert InvalidRevocation();
603             }
604         } else if (!resolver.attest{ value: value }(attestation)) {
605             revert InvalidAttestation();
606         }
607 
608         if (last) {
609             _refund(availableValue);
610         }
611 
612         return value;
613     }
```

bas-contract-main/contracts/EAS.sol #623-692

```
623     function _resolveAttestations(
624         SchemaRecord memory schemaRecord,
625         Attestation[] memory attestations,
626         uint256[] memory values,
627         bool isRevocation,
628         uint256 availableValue,
629         bool last
630     ) private returns (uint256) {
631         uint256 length = attestations.length;
632         if (length == 1) {
633             return _resolveAttestation(schemaRecord, attestations[0], values[0], isRevocation, availabl
634         }
635 
636         ISchemaResolver resolver = schemaRecord.resolver;
637         if (address(resolver) == address(0)) {
638             // Ensure that we don't accept payments if there is no resolver.
639             for (uint256 i = 0; i < length; i = uncheckedInc(i)) {
640                 if (values[i] != 0) {
641                     revert NotPayable();
642                 }
643             }
644 
645             if (last) {
646                 _refund(availableValue);
647             }
648 
649             return 0;
650         }
651 
652         uint256 totalUsedValue = 0;
653         bool isResolverPayable = resolver.isPayable();
654 
655         for (uint256 i = 0; i < length; i = uncheckedInc(i)) {
656             uint256 value = values[i];
657 
658             // Ensure that we don't accept payments which can't be forwarded to the resolver.
659             if (value == 0) {
660                 continue;
661             }
662 
663             if (!isResolverPayable) {
664                 revert NotPayable();
665             }
666 
667             // Ensure that the attester/revoker doesn't try to spend more than available.
668             if (value > availableValue) {
669                 revert InsufficientValue();
670             }
671 
672             // Ensure to deduct the sent value explicitly and add it to the total used value by the bat
673             unchecked {
674                 availableValue -= value;
675                 totalUsedValue += value;
676             }
677         }
678 
679         if (isRevocation) {
```

```
680             if (!resolver.multiRevoke{ value: totalUsedValue }(attestations, values)) {
681                 revert InvalidRevocations();
682             }
683         } else if (!resolver.multiAttest{ value: totalUsedValue }(attestations, values)) {
684             revert InvalidAttestations();
685         }
686 
687         if (last) {
688             _refund(availableValue);
689         }
690 
691         return totalUsedValue;
692     }
```

bas-contract-main/contracts/EAS.sol #717-724

```
717     function _refund(uint256 remainingValue) private {
718         if (remainingValue > 0) {
719             // Using a regular transfer here might revert, for some non-EOA attesters, due to exceedin
720             // gas limit which is why we're using call instead (via sendValue), which the 2300 gas lim
721             // apply for.
722             payable(msg.sender).sendValue(remainingValue);
723         }
724     }
```

Recommendation

Emit events in setter functions

Disclaimer

This report is governed by the stipulations (including but not limited to service descriptions, confidentiality, disclaimers, and liability limitations) outlined in the Services Agreement, or as detailed in the scope of services and terms provided to you, the Customer or Company, within the context of the Agreement. The Company is permitted to use this report only as allowed under the terms of the Agreement. Without explicit written permission from MetaTrust, this report must not be shared, disclosed, referenced, or depended upon by any third parties, nor should copies be distributed to anyone other than the Company.

It is important to clarify that this report neither endorses nor disapproves any specific project or team. It should not be viewed as a reflection of the economic value or potential of any product or asset developed by teams or projects engaging MetaTrust for security evaluations. This report does not guarantee that the technology assessed is completely free of bugs, nor does it comment on the business practices, models, or legal compliance of the technology's creators.

This report is not intended to serve as investment advice or a tool for investment decisions related to any project. It represents a thorough assessment process aimed at enhancing code quality and mitigating risks inherent in cryptographic tokens and blockchain technology. Blockchain and cryptographic assets inherently carry ongoing risks. MetaTrust's role is to support companies and individuals in their security diligence and to reduce risks associated with the use of emerging and evolving technologies. However, MetaTrust does not guarantee the security or functionality of the technologies it evaluates.

MetaTrust's assessment services are contingent on various dependencies and are continuously evolving. Accessing or using these services, including reports and materials, is at your own risk, on an as-is and as-available basis. Cryptographic tokens are novel technologies with inherent technical risks and uncertainties. The assessment reports may contain inaccuracies, such as false positives or negatives, and unpredictable outcomes. The services may rely on multiple third-party layers.

All services, labels, assessment reports, work products, and other materials, or any results from their use, are provided "as is" and "as available," with all faults and defects, without any warranty. MetaTrust expressly disclaims all warranties, whether express, implied, statutory, or otherwise, including but not limited to warranties of merchantability, fitness for a particular purpose, title, non-infringement, and any warranties arising from course of dealing, usage, or trade practice. MetaTrust does not guarantee that the services, reports, or materials will meet specific requirements, be error-free, or be compatible with other software, systems, or services.

Neither MetaTrust nor its agents make any representations or warranties regarding the accuracy, reliability, or currency of any content provided through the services. MetaTrust is not liable for any content inaccuracies, personal injuries, property damages, or any loss resulting from the use of the services, reports, or materials.

Third-party materials are provided "as is," and any warranty concerning them is strictly between the Customer and the third-party owner or distributor. The services, reports, and materials are intended solely for the Customer and should not be relied upon by others or shared without MetaTrust's consent. No third party or representative thereof shall have any rights or claims against MetaTrust regarding these services, reports, or materials.

The provisions and warranties of MetaTrust in this agreement are exclusively for the Customer's benefit. No third party has any rights or claims against MetaTrust regarding these provisions or warranties. For clarity, the services, including any assessment reports or materials, should not be used as financial, tax, legal, regulatory, or other forms of advice.