Stream: Internet Engineering Task Force (IETF)
RFC: 9175
Updates: 7252
Category: Standards Track
Published: February 2022
ISSN: 2070-1721
Authors: C. Amsüss; J. Preuß Mattsson, *Ericsson AB*; G. Selander, *Ericsson AB*

RFC 9175
Constrained Application Protocol (CoAP): Echo, Request-Tag, and Token Processing

Abstract

This document specifies enhancements to the Constrained Application Protocol (CoAP) that mitigate security issues in particular use cases. The Echo option enables a CoAP server to verify the freshness of a request or to force a client to demonstrate reachability at its claimed network address. The Request-Tag option allows the CoAP server to match block-wise message fragments belonging to the same request. This document updates RFC 7252 with respect to the following: processing requirements for client Tokens, forbidding non-secure reuse of Tokens to ensure response-to-request binding when CoAP is used with a security protocol, and amplification mitigation (where the use of the Echo option is now recommended).

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9175.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction
  1.1. Terminology
2. Request Freshness and the Echo Option
  2.1. Request Freshness
  2.2. The Echo Option
    2.2.1. Echo Option Format
  2.3. Echo Processing
  2.4. Applications of the Echo Option
  2.5. Characterization of Echo Applications
    2.5.1. Time-Based versus Event-Based Freshness
    2.5.2. Authority over Used Information
    2.5.3. Protection by a Security Protocol
  2.6. Updated Amplification Mitigation Requirements for Servers
3. Protecting Message Bodies Using Request Tags
  3.1. Fragmented Message Body Integrity
  3.2. The Request-Tag Option
    3.2.1. Request-Tag Option Format
  3.3. Request-Tag Processing by Servers
  3.4. Setting the Request-Tag
  3.5. Applications of the Request-Tag Option
    3.5.1. Body Integrity Based on Payload Integrity
    3.5.2. Multiple Concurrent Block-Wise Operations
    3.5.3. Simplified Block-Wise Handling for Constrained Proxies

3.6. Rationale for the Option Properties

3.7. Rationale for Introducing the Option

3.8. Block2 and ETag Processing

4. Token Processing for Secure Request-Response Binding

4.1. Request-Response Binding

4.2. Updated Token Processing Requirements for Clients

5. Security Considerations

5.1. Token Reuse

6. Privacy Considerations

7. IANA Considerations

8. References

8.1. Normative References

8.2. Informative References

Appendix A. Methods for Generating Echo Option Values

Appendix B. Request-Tag Message Size Impact

Acknowledgements

Authors' Addresses

1. Introduction

The initial suite of specifications for the Constrained Application Protocol (CoAP) ([RFC7252], [RFC7641], and [RFC7959]) was designed with the assumption that security could be provided on a separate layer, in particular, by using DTLS [RFC6347]. However, for some use cases, additional functionality or extra processing is needed to support secure CoAP operations. This document specifies security enhancements to CoAP.

This document specifies two CoAP options, the Echo option and the Request-Tag option. The Echo option enables a CoAP server to verify the freshness of a request, which can be used to synchronize state, or to force a client to demonstrate reachability at its claimed network address. The Request-Tag option allows the CoAP server to match message fragments belonging to the same request, fragmented using the CoAP block-wise transfer mechanism, which mitigates attacks and enables concurrent block-wise operations. These options in themselves do not replace the need for a security protocol; they specify the format and processing of data that, when integrity protected using, e.g., DTLS [RFC6347], TLS [RFC8446], or Object Security for Constrained RESTful Environments (OSCORE) [RFC8613], provide the additional security features.

This document updates [RFC7252] with a recommendation that servers use the Echo option to mitigate amplification attacks.

The document also updates the Token processing requirements for clients specified in [RFC7252]. The updated processing forbids non-secure reuse of Tokens to ensure binding of responses to requests when CoAP is used with security, thus mitigating error cases and attacks where the client may erroneously associate the wrong response to a request.

Each of the following sections provides a more-detailed introduction to the topic at hand in its first subsection.

1.1. Terminology

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Like [RFC7252], this document relies on the Representational State Transfer [REST] architecture of the Web.

Unless otherwise specified, the terms "client" and "server" refer to "CoAP client" and "CoAP server", respectively, as defined in [RFC7252].

A message's "freshness" is a measure of when a message was sent on a timescale of the recipient. A server that receives a request can either verify that the request is fresh or determine that it cannot be verified that the request is fresh. What is considered a fresh message is application dependent; exemplary uses are "no more than 42 seconds ago" or "after this server's last reboot".

The terms "payload" and "body" of a message are used as in [RFC7959]. The complete interchange of a request and a response body is called a (REST) "operation". An operation fragmented using [RFC7959] is called a "block-wise operation". A block-wise operation that is fragmenting the request body is called a "block-wise request operation". A block-wise operation that is fragmenting the response body is called a "block-wise response operation".

Two request messages are said to be "matchable" if they occur between the same endpoint pair, have the same code, and have the same set of options, with the exception that elective NoCacheKey options and options involved in block-wise transfer (Block1, Block2, and Request-Tag) need not be the same. Two blockwise request operations are said to be matchable if their request messages are matchable.

Two matchable block-wise request operations are said to be "concurrent" if a block of the second request is exchanged even though the client still intends to exchange further blocks in the first operation. (Concurrent block-wise request operations from a single endpoint are impossible with the options of [RFC7959] -- see the last paragraphs of Sections 2.4 and 2.5 -- because the second operation's block overwrites any state of the first exchange.)

The Echo and Request-Tag options are defined in this document.

2. Request Freshness and the Echo Option

2.1. Request Freshness

A CoAP server receiving a request is, in general, not able to verify when the request was sent by the CoAP client. This remains true even if the request was protected with a security protocol, such as DTLS. This makes CoAP requests vulnerable to certain delay attacks that are particularly perilous in the case of actuators [COAP-ATTACKS]. Some attacks can be mitigated by establishing fresh session keys, e.g., performing a DTLS handshake for each request, but, in general, this is not a solution suitable for constrained environments, for example, due to increased message overhead and latency. Additionally, if there are proxies, fresh DTLS session keys between the server and the proxy do not say anything about when the client made the request. In a general hop-by-hop setting, freshness may need to be verified in each hop.

A straightforward mitigation of potential delayed requests is that the CoAP server rejects a request the first time it appears and asks the CoAP client to prove that it intended to make the request at this point in time.

2.2. The Echo Option

This document defines the Echo option, a lightweight challenge-response mechanism for CoAP that enables a CoAP server to verify the freshness of a request. A fresh request is one whose age has not yet exceeded the freshness requirements set by the server. The freshness requirements are application specific and may vary based on resource, method, and parameters outside of CoAP, such as policies. The Echo option value is a challenge from the server to the client included in a CoAP response and echoed back to the server in one or more CoAP requests.

This mechanism is not only important in the case of actuators, or other use cases where the CoAP operations require freshness of requests, but also in general for synchronizing state between a CoAP client and server, cryptographically verifying the aliveness of the client or forcing a client to demonstrate reachability at its claimed network address. The same functionality can be provided by echoing freshness indicators in CoAP payloads, but this only works for methods and response codes defined to have a payload. The Echo option provides a convention to transfer freshness indicators that works for all methods and response codes.

2.2.1. Echo Option Format

The Echo option is elective, safe to forward, not part of the cache-key, and not repeatable (see Table 1, which extends Table 4 of [RFC7252]).

No.	C	U	N	R	Name	Format	Length	Default
252			x		Echo	opaque	1-40	(none)

Table 1: Echo Option Summary

C=Critical, U=Unsafe, N=NoCacheKey, R=Repeatable

The Echo option value is generated by a server, and its content and structure are implementation specific. Different methods for generating Echo option values are outlined in Appendix A. Clients and intermediaries **MUST** treat an Echo option value as opaque and make no assumptions about its content or structure.

When receiving an Echo option in a request, the server **MUST** be able to verify that the Echo option value (a) was generated by the server or some other party that the server trusts and (b) fulfills the freshness requirements of the application. Depending on the freshness requirements, the server may verify exactly when the Echo option value was generated (time-based freshness) or verify that the Echo option was generated after a specific event (event-based freshness). As the request is bound to the Echo option value, the server can determine that the request is not older than the Echo option value.

When the Echo option is used with OSCORE [RFC8613], it **MAY** be an Inner or Outer option, and the Inner and Outer values are independent. OSCORE servers **MUST** only produce Inner Echo options unless they are merely testing for reachability of the client (the same as proxies may do). The Inner option is encrypted and integrity protected between the endpoints, whereas the Outer option is not protected by OSCORE. As always with OSCORE, Outer options are visible to (and may be acted on by) all proxies and are visible on all links where no additional encryption (like TLS between client and proxy) is used.

2.3. Echo Processing

The Echo option **MAY** be included in any request or response (see Section 2.4 for different applications).

The application decides under what conditions a CoAP request to a resource is required to be fresh. These conditions can, for example, include what resource is requested, the request method and other data in the request, and conditions in the environment, such as the state of the server or the time of the day.

If a certain request is required to be fresh, the request does not contain a fresh Echo option value, and the server cannot verify the freshness of the request in some other way, the server **MUST NOT** process the request further and **SHOULD** send a 4.01 (Unauthorized) response with an Echo option. The server **MAY** include the same Echo option value in several different response messages and to different clients. Examples of this could be time-based freshness (when several responses are sent closely after each other) or event-based freshness (with no event taking place between the responses).

The server may use request freshness provided by the Echo option to verify the aliveness of a client or to synchronize state. The server may also include the Echo option in a response to force a client to demonstrate reachability at its claimed network address. Note that the Echo option does not bind a request to any particular previous response but provides an indication that the client had access to the previous response at the time when it created the request.

Upon receiving a 4.01 (Unauthorized) response with the Echo option, the client **SHOULD** resend the original request with the addition of an Echo option with the received Echo option value. The client **MAY** send a different request compared to the original request. Upon receiving any other response with the Echo option, the client **SHOULD** echo the Echo option value in the next request to the server. The client **MAY** include the same Echo option value in several different requests to the server or discard it at any time (especially to avoid tracking; see Section 6).

A client **MUST** only send Echo option values to endpoints it received them from (where, as defined in Section 1.2 of [RFC7252], the security association is part of the endpoint). In OSCORE processing, that means sending Echo option values from Outer options (or from non-OSCORE responses) back in Outer options and sending those from Inner options in Inner options in the same security context.

Upon receiving a request with the Echo option, the server determines if the request is required to be fresh. If not, the Echo option **MAY** be ignored. If the request is required to be fresh and the server cannot verify the freshness of the request in some other way, the server **MUST** use the Echo option to verify that the request is fresh. If the server cannot verify that the request is fresh, the request is not processed further, and an error message **MAY** be sent. The error message **SHOULD** include a new Echo option.

One way for the server to verify freshness is to bind the Echo option value to a specific point in time and verify that the request is not older than a certain threshold T. The server can verify this by checking that (t1 - t0) < T, where t1 is the request receive time and t0 is the time when the Echo option value was generated. An example message flow over DTLS is shown Figure 1.

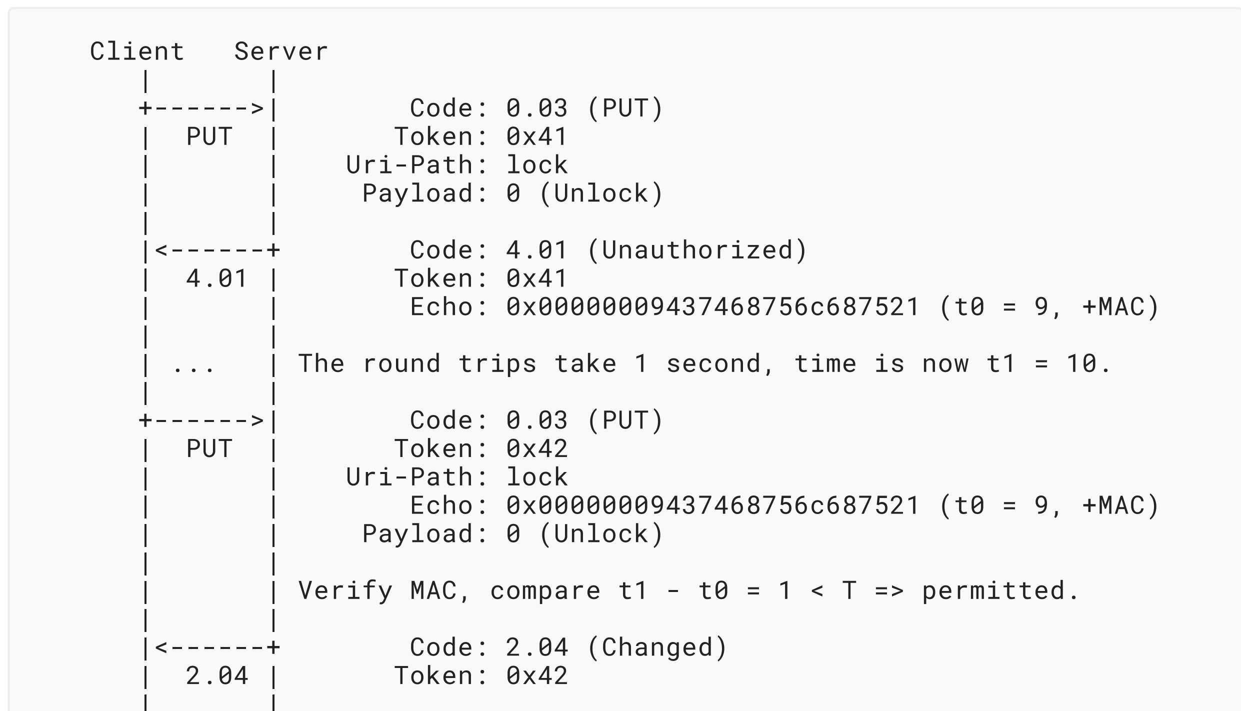
```
   Client   Server
      |        |
      +------->|        Code: 0.03 (PUT)
      |  PUT   |       Token: 0x41
      |        |    Uri-Path: lock
      |        |     Payload: 0 (Unlock)
      |        |
      |<-------+        Code: 4.01 (Unauthorized)
      |  4.01  |       Token: 0x41
      |        |        Echo: 0x00000009437468756c687521 (t0 = 9, +MAC)
      |        |
      | ...    | The round trips take 1 second, time is now t1 = 10.
      |        |
      +------->|        Code: 0.03 (PUT)
      |  PUT   |       Token: 0x42
      |        |    Uri-Path: lock
      |        |        Echo: 0x00000009437468756c687521 (t0 = 9, +MAC)
      |        |     Payload: 0 (Unlock)
      |        |
      |        | Verify MAC, compare t1 - t0 = 1 < T => permitted.
      |        |
      |<-------+        Code: 2.04 (Changed)
      |  2.04  |       Token: 0x42
      |        |
```

Figure 1: Example Message Flow for Time-Based Freshness Using the 'Integrity-Protected Timestamp' Construction of Appendix A

Another way for the server to verify freshness is to maintain a cache of values associated to events. The size of the cache is defined by the application. In the following, we assume the cache size is 1, in which case, freshness is defined as "no new event has taken place". At each event, a new value is written into the cache. The cache values **MUST** be different or chosen in a way so the probability for collisions is negligible. The server verifies freshness by checking that e0 equals e1, where e0 is the cached value when the Echo option value was generated, and e1 is the cached value at the reception of the request. An example message flow over DTLS is shown in Figure 2.

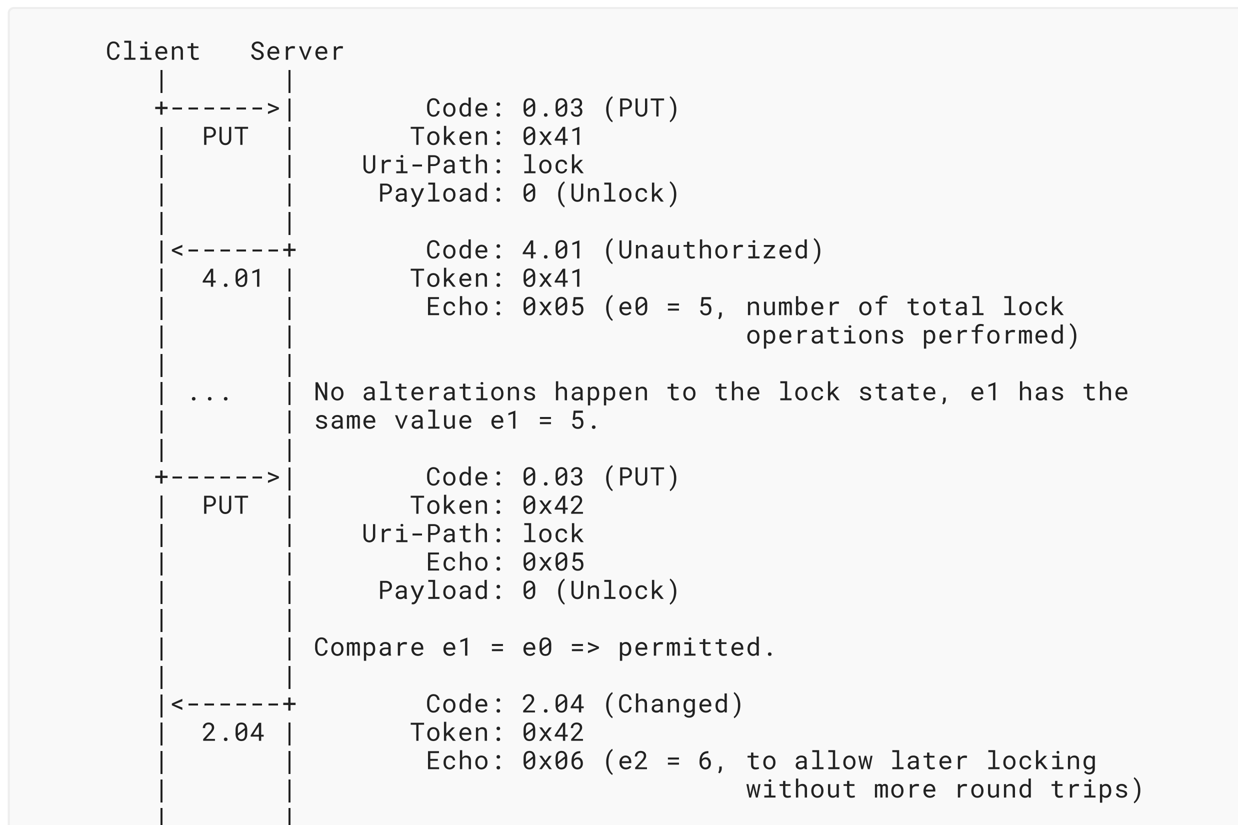

Figure 2: Example Message Flow for Event-Based Freshness Using the 'Persistent Counter' Construction of Appendix A

When used to serve freshness requirements (including client aliveness and state synchronizing), the Echo option value **MUST** be integrity protected between the intended endpoints, e.g., using DTLS, TLS, or an OSCORE Inner option [RFC8613]. When used to demonstrate reachability at a claimed network address, the Echo option **SHOULD** be a Message Authentication Code (MAC) of the claimed address but **MAY** be unprotected. Combining different Echo applications can necessitate different choices; see Appendix A, item 2 for an example.

An Echo option **MAY** be sent with a successful response, i.e., even though the request satisfied any freshness requirements on the operation. This is called a "preemptive" Echo option value and is useful when the server anticipates that the client will need to demonstrate freshness relative to the current response in the near future.

A CoAP-to-CoAP proxy **MAY** set an Echo option on responses, both on forwarded ones that had no Echo option or ones generated by the proxy (from cache or as an error). If it does so, it **MUST** remove the Echo option it recognizes as one generated by itself on follow-up requests. When it receives an Echo option in a response, it **MAY** forward it to the client (and, not recognizing it as its own in future requests, relay it in the other direction as well) or process it on its own. If it does so, it **MUST** ensure that the client's request was generated (or is regenerated) after the Echo option value used to send to the server was first seen. (In most cases, this means that the proxy needs to ask the client to repeat the request with a new Echo option value.)

The CoAP server side of CoAP-to-HTTP proxies **MAY** request freshness, especially if they have reason to assume that access may require it (e.g., because it is a PUT or POST); how this is determined is out of scope for this document. The CoAP client side of HTTP-to-CoAP proxies **MUST** respond to Echo challenges itself if the proxy knows from the recent establishing of the connection that the HTTP request is fresh. Otherwise, it **MUST NOT** repeat an unsafe request and **SHOULD** respond with a 503 (Service Unavailable) with a Retry-After value of 0 seconds and terminate any underlying Keep-Alive connection. If the HTTP request arrived in early data, the proxy **SHOULD** use a 425 (Too Early) response instead (see [RFC8470]). The proxy **MAY** also use other mechanisms to establish freshness of the HTTP request that are not specified here.

2.4. Applications of the Echo Option

Unless otherwise noted, all these applications require a security protocol to be used and the Echo option to be protected by it.

1. Actuation requests often require freshness guarantees to avoid accidental or malicious delayed actuator actions. In general, all unsafe methods (e.g., POST, PUT, and DELETE) may require freshness guarantees for secure operation.
   - The same Echo option value may be used for multiple actuation requests to the same server, as long as the total time since the Echo option value was generated is below the freshness threshold.
   - For actuator applications with low delay tolerance, to avoid additional round trips for multiple requests in rapid sequence, the server may send preemptive Echo option values in successful requests, irrespectively of whether or not the request contained an Echo option. The client then uses the Echo option with the new value in the next actuation request, and the server compares the receive time accordingly.

2. A server may use the Echo option to synchronize properties (such as state or time) with a requesting client. A server **MUST NOT** synchronize a property with a client that is not the authority of the property being synchronized. For example, if access to a server resource is dependent on time, then the server **MUST NOT** synchronize time with a client requesting access unless the client is a time authority for the server.

   Note that the state to be synchronized is not carried inside the Echo option. Any explicit state information needs to be carried along in the messages the Echo option value is sent in; the Echo mechanism only provides a partial order on the messages' processing.
   - If a server reboots during operation, it may need to synchronize state or time before continuing the interaction. For example, with OSCORE, it is possible to reuse a partly persistently stored security context by synchronizing the Partial IV (sequence number) using the Echo option, as specified in Section 7.5 of [RFC8613].
   - A device joining a CoAP group communication [GROUP-COAP] protected with OSCORE [GROUP-OSCORE] may be required to initially synchronize its replay window state with a client by using the Echo option in a unicast response to a multicast request. The client receiving the response with the Echo option includes the Echo option value in a subsequent unicast request to the responding server.

3. An attacker can perform a denial-of-service attack by putting a victim's address in the source address of a CoAP request and sending the request to a resource with a large amplification factor. The amplification factor is the ratio between the size of the request and the total size of the response(s) to that request. A server that provides a large amplification factor to an unauthenticated peer **SHOULD** mitigate amplification attacks, as described in Section 11.3 of [RFC7252]. One way to mitigate such attacks is for the server to respond to the alleged source address of the request with an Echo option in a short response message (e.g., 4.01 (Unauthorized)), thereby requesting the client to verify its source address. This needs to be done only once per endpoint and limits the range of potential victims from the general Internet to endpoints that have been previously in contact with the server. For this application, the Echo option can be used in messages that are not integrity protected, for example, during discovery. (This is formally recommended in Section 2.6.)
   - In the presence of a proxy, a server will not be able to distinguish different origin client endpoints, i.e., the client from which a request originates. Following from the recommendation above, a proxy that provides a large amplification factor to unauthenticated peers **SHOULD** mitigate amplification attacks. The proxy **SHOULD** use the Echo option to verify origin reachability, as described in Section 2.3. The proxy **MAY** forward safe requests immediately to have a cached result available when the client's repeated request arrives.
   - Amplification mitigation is a trade-off between giving leverage to an attacker and causing overhead. An amplification factor of 3 (i.e., don't send more than three times the number of bytes received until the peer's address is confirmed) is considered acceptable for unconstrained applications in [RFC9000], Section 8.

     When that limit is applied and no further context is available, a safe default is sending initial responses no larger than 136 bytes in CoAP serialization. (The number is assuming Ethernet, IP, and UDP headers of 14, 40, and 8 bytes, respectively, with 4 bytes added for the CoAP header. Triple that minus the non-CoAP headers gives the 136 bytes.) Given the token also takes up space in the request, responding with 132 bytes after the token is safe as well.
   - When an Echo response is sent to mitigate amplification, it **MUST** be sent as a piggybacked or Non-confirmable response, never as a separate one (which would cause amplification due to retransmission).

4. A server may want to use the request freshness provided by the Echo option to verify the aliveness of a client. Note that, in a deployment with hop-by-hop security and proxies, the server can only verify aliveness of the closest proxy.

2.5. Characterization of Echo Applications

Use cases for the Echo option can be characterized by several criteria that help determine the required properties of the Echo option value. These criteria apply both to those listed in Section 2.4 and any novel applications. They provide rationale for the statements in the former and guidance for the latter.

2.5.1. Time-Based versus Event-Based Freshness

The property a client demonstrates by sending an Echo option value is that the request was sent after a certain point in time or after some event happened on the server.

When events are counted, they form something that can be used as a monotonic but very non-uniform time line. With highly regular events and low-resolution time, the distinction between time-based and event-based freshness can be blurred: "no longer than a month ago" is similar to "since the last full moon".

In an extreme form of event-based freshness, the server can place an event whenever an Echo option value is used. This makes the Echo option value effectively single use.

Event-based and time-based freshness can be combined in a single Echo option value, e.g., by encrypting a timestamp with a key that changes with every event to obtain semantics in the style of "usable once but only for 5 minutes".

2.5.2. Authority over Used Information

Information conveyed to the server in the request Echo option value has different authority depending on the application. Understanding who or what is the authoritative source of that information helps the server implementor decide the necessary protection of the Echo option value.

If all that is conveyed to the server is information that the client is authorized to provide arbitrarily (which is another way of saying that the server has to trust the client on whatever the Echo option is being used for), then the server can issue Echo option values that do not need to be protected on their own. They still need to be covered by the security protocol that covers the rest of the message, but the Echo option value can be just short enough to be unique between this server and client.

For example, the client's OSCORE Sender Sequence Number (as used in [RFC8613], Appendix B.1.2) is such information.

In most other cases, there is information conveyed for which the server is the authority ("the request must not be older than five minutes" is counted on the server's clock, not the client's) or which even involve the network (as when performing amplification mitigation). In these cases, the Echo option value itself needs to be protected against forgery by the client, e.g., by using a sufficiently large, random value or a MAC, as described in Appendix A, items 1 and 2.

For some applications, the server may be able to trust the client to also act as the authority (e.g., when using time-based freshness purely to mitigate request delay attacks); these need careful case-by-case evaluation.

To issue Echo option values without integrity protection of its own, the server needs to trust the client to never produce requests with attacker-controlled Echo option values. The provisions of Section 2.3 (saying that an Echo option value may only be sent as received from the same server) allow that. The requirement stated there for the client to treat the Echo option value as opaque holds for these applications like for all others.

When the client is the sole authority over the synchronized property, the server can still use time or events to issue new Echo option values. Then, the request's Echo option value not so much proves the indicated freshness to the server but reflects the client's intention to indicate reception of responses containing that value when sending the later ones.

Note that a single Echo option value can be used for multiple purposes (e.g., to both get the sequence number information and perform amplification mitigation). In this case, the stricter protection requirements apply.

2.5.3. Protection by a Security Protocol

For meaningful results, the Echo option needs to be used in combination with a security protocol in almost all applications.

When the information extracted by the server is only about a part of the system outside of any security protocol, then the Echo option can also be used without a security protocol (in case of OSCORE, as an Outer option).

The only known application satisfying this requirement is network address reachability, where unprotected Echo option values are used both by servers (e.g., during setup of a security context) and proxies (which do not necessarily have a security association with their clients) for amplification mitigation.

2.6. Updated Amplification Mitigation Requirements for Servers

This section updates the amplification mitigation requirements for servers in [RFC7252] to recommend the use of the Echo option to mitigate amplification attacks. The requirements for clients are not updated. Section 11.3 of [RFC7252] is updated by adding the following text:

> A CoAP server **SHOULD** mitigate potential amplification attacks by responding to unauthenticated clients with 4.01 (Unauthorized) including an Echo option, as described in item 3 in Section 2.4 of RFC 9175.

3. Protecting Message Bodies Using Request Tags

3.1. Fragmented Message Body Integrity

CoAP was designed to work over unreliable transports, such as UDP, and includes a lightweight reliability feature to handle messages that are lost or arrive out of order. In order for a security protocol to support CoAP operations over unreliable transports, it must allow out-of-order delivery of messages.

The block-wise transfer mechanism [RFC7959] extends CoAP by defining the transfer of a large resource representation (CoAP message body) as a sequence of blocks (CoAP message payloads). The mechanism uses a pair of CoAP options, Block1 and Block2, pertaining to the request and response payload, respectively. The block-wise functionality does not support the detection of interchanged blocks between different message bodies to the same resource having the same block number. This remains true even when CoAP is used together with a security protocol (such as DTLS or OSCORE) within the replay window [COAP-ATTACKS], which is a vulnerability of the block-wise functionality of CoAP [RFC7959].

A straightforward mitigation of mixing up blocks from different messages is to use unique identifiers for different message bodies, which would provide equivalent protection to the case where the complete body fits into a single payload. The ETag option [RFC7252], set by the CoAP server, identifies a response body fragmented using the Block2 option.

3.2. The Request-Tag Option

This document defines the Request-Tag option for identifying request bodies, similar to ETag, but ephemeral and set by the CoAP client. The Request-Tag is intended for use as a short-lived identifier for keeping apart distinct block-wise request operations on one resource from one client, addressing the issue described in Section 3.1. It enables the receiving server to reliably assemble request payloads (blocks) to their message bodies and, if it chooses to support it, to reliably process simultaneous block-wise request operations on a single resource. The requests must be integrity protected if they should protect against interchange of blocks between different message bodies. The Request-Tag option is mainly used in requests that carry the Block1 option and in Block2 requests following these.

In essence, it is an implementation of the "proxy-safe elective option" used just to "vary the cache key", as suggested in [RFC7959], Section 2.4.

3.2.1. Request-Tag Option Format

The Request-Tag option is elective, safe to forward, repeatable, and part of the cache key (see Table 2, which extends Table 4 of [RFC7252]).

No.	C	U	N	R	Name	Format	Length	Default
292				x	Request-Tag	opaque	0-8	(none)

Table 2: Request-Tag Option Summary

C=Critical, U=Unsafe, N=NoCacheKey, R=Repeatable

Request-Tag, like the Block options, is both a class E and a class U option in terms of OSCORE processing (see Section 4.1 of [RFC8613]). The Request-Tag **MAY** be an Inner or Outer option. It influences the Inner or Outer block operations, respectively. The Inner and Outer values are therefore independent of each other. The Inner option is encrypted and integrity protected between the client and server, and it provides message body identification in case of end-to-end fragmentation of requests. The Outer option is visible to proxies and labels message bodies in case of hop-by-hop fragmentation of requests.

The Request-Tag option is only used in the request messages of block-wise operations.

The Request-Tag mechanism can be applied independently on the server and client sides of CoAP-to-CoAP proxies, as are the Block options. However, given it is safe to forward, a proxy is free to just forward it when processing an operation. CoAP-to-HTTP proxies and HTTP-to-CoAP proxies can use Request-Tag on their CoAP sides; it is not applicable to HTTP requests.

3.3. Request-Tag Processing by Servers

The Request-Tag option does not require any particular processing on the server side outside of the processing already necessary for any unknown elective proxy-safe cache-key option. The option varies the properties that distinguish block-wise operations (which includes all options except Block1, Block2, and all operations that are elective NoCacheKey). Thus, the server cannot treat messages with a different list of Request-Tag options as belonging to the same operation.

To keep utilizing the cache, a server (including proxies) **MAY** discard the Request-Tag option from an assembled block-wise request when consulting its cache, as the option relates to the operation on the wire and not its semantics. For example, a FETCH request with the same body as an older one can be served from the cache if the older's Max-Age has not expired yet, even if the second operation uses a Request-Tag and the first did not. (This is similar to the situation about ETag in that it is formally part of the cache key, but implementations that are aware of its meaning can cache more efficiently (see [RFC7252], Section 5.4.2).

A server receiving a Request-Tag **MUST** treat it as opaque and make no assumptions about its content or structure.

Two messages carrying the same Request-Tag is a necessary but not sufficient condition for being part of the same operation. For one, a server may still treat them as independent messages when it sends 2.01 (Created) and 2.04 (Changed) responses for every block. Also, a client that lost interest in an old operation but wants to start over can overwrite the server's old state with a new initial (num=0) Block1 request and the same Request-Tag under some circumstances. Likewise, that results in the new message not being part of the old operation.

As it has always been, a server that can only serve a limited number of block-wise operations at the same time can delay the start of the operation by replying with 5.03 (Service Unavailable) and a Max-Age indicating how long it expects the existing operation to go on, or it can forget about the state established with the older operation and respond with 4.08 (Request Entity Incomplete) to later blocks on the first operation.

3.4. Setting the Request-Tag

For each separate block-wise request operation, the client can choose a Request-Tag value or choose not to set a Request-Tag. It needs to be set to the same value (or unset) in all messages belonging to the same operation; otherwise, they are treated as separate operations by the server.

Starting a request operation matchable to a previous operation and even using the same Request-Tag value is called "request tag recycling". The absence of a Request-Tag option is viewed as a value distinct from all values with a single Request-Tag option set; starting a request operation matchable to a previous operation where neither has a Request-Tag option therefore constitutes request tag recycling just as well (also called "recycling the absent option").

Clients that use Request-Tag for a particular purpose (like in Section 3.5) **MUST NOT** recycle a request tag unless the first operation has concluded. What constitutes a concluded operation depends on the purpose and is defined accordingly; see examples in Section 3.5.

When Block1 and Block2 are combined in an operation, the Request-Tag of the Block1 phase is set in the Block2 phase as well; otherwise, the request would have a different set of options and would not be recognized any more.

Clients are encouraged to generate compact messages. This means sending messages without Request-Tag options whenever possible and using short values when the absent option cannot be recycled.

Note that Request-Tag options can be present in request messages that carry no Block options (for example, because a proxy unaware of Request-Tag reassembled them).

The Request-Tag option **MUST NOT** be present in response messages.

3.5. Applications of the Request-Tag Option

3.5.1. Body Integrity Based on Payload Integrity

When a client fragments a request body into multiple message payloads, even if the individual messages are integrity protected, it is still possible for an attacker to maliciously replace a later operation's blocks with an earlier operation's blocks (see Section 2.5 of [COAP-ATTACKS]). Therefore, the integrity protection of each block does not extend to the operation's request body.

In order to gain that protection, use the Request-Tag mechanism as follows:

- The individual exchanges **MUST** be integrity protected end to end between the client and server.

- The client **MUST NOT** recycle a request tag in a new operation unless the previous operation matchable to the new one has concluded.

  If any future security mechanisms allow a block-wise transfer to continue after an endpoint's details (like the IP address) have changed, then the client **MUST** consider messages matchable if they were sent to any endpoint address using the new operation's security context.
- The client **MUST NOT** regard a block-wise request operation as concluded unless all of the messages the client has sent in the operation would be regarded as invalid by the server if they were replayed.

  When security services are provided by OSCORE, these confirmations typically result either from the client receiving an OSCORE response message matching the request (an empty Acknowledgement (ACK) is insufficient) or because the message's sequence number is old enough to be outside the server's receive window.

  When security services are provided by DTLS, this can only be confirmed if there was no CoAP retransmission of the request, the request was responded to, and the server uses replay protection.

Authors of other documents (e.g., applications of [RFC8613]) are invited to mandate this subsection's behavior for clients that execute block-wise interactions over secured transports. In this way, the server can rely on a conforming client to set the Request-Tag option when required and thereby have confidence in the integrity of the assembled body.

Note that this mechanism is implicitly implemented when the security layer guarantees ordered delivery (e.g., CoAP over TLS [RFC8323]). This is because, with each message, any earlier message cannot be replayed any more, so the client never needs to set the Request-Tag option unless it wants to perform concurrent operations.

Body integrity only makes sense in applications that have stateful block-wise transfers. On applications where all the state is in the application (e.g., because rather than POSTing a large representation to a collection in a stateful block-wise transfer, a collection item is created first, then written to once and available when written completely), clients need not concern themselves with body integrity and thus the Request-Tag.

Body integrity is largely independent from replay protection. When no replay protection is available (it is optional in DTLS), a full block-wise operation may be replayed, but, by adhering to the above, no operations will be mixed up. The only link between body integrity and replay protection is that, without replay protection, recycling is not possible.

3.5.2. Multiple Concurrent Block-Wise Operations

CoAP clients, especially CoAP proxies, may initiate a block-wise request operation to a resource, to which a previous one is already in progress, which the new request should not cancel. A CoAP proxy would be in such a situation when it forwards operations with the same cache-key options but possibly different payloads.

For those cases, Request-Tag is the proxy-safe elective option suggested in the last paragraph of Section 2.4 of [RFC7959].

When initializing a new block-wise operation, a client has to look at other active operations:

- If any of them is matchable to the new one, and the client neither wants to cancel the old one nor postpone the new one, it can pick a Request-Tag value (including the absent option) that is not in use by the other matchable operations for the new operation.
- Otherwise, it can start the new operation without setting the Request-Tag option on it.

3.5.3. Simplified Block-Wise Handling for Constrained Proxies

The Block options were defined to be unsafe to forward because a proxy that would forward blocks as plain messages would risk mixing up clients' requests.

In some cases, for example, when forwarding block-wise request operations, appending a Request-Tag value unique to the client can satisfy the requirements on the proxy that come from the presence of a Block option.

This is particularly useful to proxies that strive for stateless operations, as described in [RFC8974], Section 4.

The precise classification of cases in which such a Request-Tag option is sufficient is not trivial, especially when both request and response body are fragmented, and is out of scope for this document.

3.6. Rationale for the Option Properties

The Request-Tag option can be elective, because to servers unaware of the Request-Tag option, operations with differing request tags will not be matchable.

The Request-Tag option can be safe to forward but part of the cache key, because proxies unaware of the Request-Tag option will consider operations with differing request tags unmatchable but can still forward them.

The Request-Tag option is repeatable because this easily allows several cascaded stateless proxies to each put in an origin address. They can perform the steps of Section 3.5.3 without the need to create an option value that is the concatenation of the received option and their own value and can simply add a new Request-Tag option unconditionally.

In draft versions of this document, the Request-Tag option used to be critical and unsafe to forward. That design was based on an erroneous understanding of which blocks could be composed according to [RFC7959].

3.7. Rationale for Introducing the Option

An alternative that was considered to the Request-Tag option for coping with the problem of fragmented message body integrity (Section 3.5.1) was to update [RFC7959] to say that blocks could only be assembled if their fragments' order corresponded to the sequence numbers.

That approach would have been difficult to roll out reliably on DTLS, where many implementations do not expose sequence numbers, and would still not prevent attacks like in Section 2.5.2 of [COAP-ATTACKS].

3.8. Block2 and ETag Processing

The same security properties as in Section 3.5.1 can be obtained for block-wise response operations. The threat model here does not depend on an attacker; a client can construct a wrong representation by assembling it from blocks from different resource states. That can happen when a resource is modified during a transfer or when some blocks are still valid in the client's cache.

Rules stating that response body reassembly is conditional on matching ETag values are already in place from Section 2.4 of [RFC7959].

To gain protection equivalent to that described in Section 3.5.1, a server **MUST** use the Block2 option in conjunction with the ETag option ([RFC7252], Section 5.10.6) and **MUST NOT** use the same ETag value for different representations of a resource.

4. Token Processing for Secure Request-Response Binding

4.1. Request-Response Binding

A fundamental requirement of secure REST operations is that the client can bind a response to a particular request. If this is not ensured, a client may erroneously associate the wrong response to a request. The wrong response may be an old response for the same resource or a response for a completely different resource (e.g., see Section 2.3 of [COAP-ATTACKS]). For example, a request for the alarm status "GET /status" may be associated to a prior response "on", instead of the correct response "off".

In HTTP/1.1, this type of binding is always assured by the ordered and reliable delivery, as well as mandating that the server sends responses in the same order that the requests were received. The same is not true for CoAP, where the server (or an attacker) can return responses in any order and where there can be any number of responses to a request (e.g., see [RFC7641]). In CoAP, concurrent requests are differentiated by their Token. Note that the CoAP Message ID cannot be used for this purpose since those are typically different for the REST request and corresponding response in case of "separate response" (see Section 2.2 of [RFC7252]).

CoAP [RFC7252] does not treat the Token as a cryptographically important value and does not give stricter guidelines than that the Tokens currently "in use" **SHOULD** (not **SHALL**) be unique. If used with a security protocol not providing bindings between requests and responses (e.g., DTLS and TLS), Token reuse may result in situations where a client matches a response to the wrong request. Note that mismatches can also happen for other reasons than a malicious attacker, e.g., delayed delivery or a server sending notifications to an uninterested client.

A straightforward mitigation is to mandate clients to not reuse Tokens until the traffic keys have been replaced. The following section formalizes that.

4.2. Updated Token Processing Requirements for Clients

As described in Section 4.1, the client must be able to verify that a response corresponds to a particular request. This section updates the Token processing requirements for clients in [RFC7252] to always assure a cryptographically secure binding of responses to requests for secure REST operations like "coaps". The Token processing for servers is not updated. Token processing in Section 5.3.1 of [RFC7252] is updated by adding the following text:

> When CoAP is used with a security protocol not providing bindings between requests and responses, the Tokens have cryptographic importance. The client **MUST** make sure that Tokens are not used in a way so that responses risk being associated with the wrong request.
>
> One easy way to accomplish this is to implement the Token (or part of the Token) as a sequence number, starting at zero for each new or rekeyed secure connection. This approach **SHOULD** be followed.

5. Security Considerations

The freshness assertion of the Echo option comes from the client reproducing the same value of the Echo option in a request as it received in a previous response. If the Echo option value is a large random number, then there is a high probability that the request is generated after having seen the response. If the Echo option value of the response can be guessed, e.g., if based on a small random number or a counter (see Appendix A), then it is possible to compose a request with the right Echo option value ahead of time. Using guessable Echo option values is only permissible in a narrow set of cases described in Section 2.5.2. Echo option values **MUST** be set by the CoAP server such that the risk associated with unintended reuse can be managed.

If uniqueness of the Echo option value is based on randomness, then the availability of a secure pseudorandom number generator and truly random seeds are essential for the security of the Echo option. If no true random number generator is available, a truly random seed must be provided from an external source. As each pseudorandom number must only be used once, an implementation needs to get a new truly random seed after reboot or continuously store the state in nonvolatile memory. See [RFC8613], Appendix B.1.1 for issues and approaches for writing to nonvolatile memory.

A single active Echo option value with 64 (pseudo)random bits gives the same theoretical security level as a 64-bit MAC (as used in, e.g., AES_128_CCM_8). If a random unique Echo option value is intended, the Echo option value **SHOULD** contain 64 (pseudo)random bits that are not predictable for any other party than the server. A server **MAY** use different security levels for different use cases (client aliveness, request freshness, state synchronization, network address reachability, etc.).

The security provided by the Echo and Request-Tag options depends on the security protocol used. CoAP and HTTP proxies require (D)TLS to be terminated at the proxies. The proxies are therefore able to manipulate, inject, delete, or reorder options or packets. The security claims in such architectures only hold under the assumption that all intermediaries are fully trusted and have not been compromised.

Echo option values without the protection of randomness or a MAC are limited to cases when the client is the trusted source of all derived properties (as per Section 2.5.2). Using them needs per-application consideration of both the impact of a malicious client and of implementation errors in clients. These Echo option values are the only legitimate case for Echo option values shorter than four bytes, which are not necessarily secret. They **MUST NOT** be used unless the Echo option values in the request are integrity protected, as per Section 2.3.

Servers **SHOULD** use a monotonic clock to generate timestamps and compute round-trip times. Use of non-monotonic clocks is not secure, as the server will accept expired Echo option values if the clock is moved backward. The server will also reject fresh Echo option values if the clock is moved forward. Non-monotonic clocks **MAY** be used as long as they have deviations that are acceptable given the freshness requirements. If the deviations from a monotonic clock are known, it may be possible to adjust the threshold accordingly.

An attacker may be able to affect the server's system time in various ways, such as setting up a fake NTP server or broadcasting false time signals to radio-controlled clocks.

For the purpose of generating timestamps for the Echo option, a server **MAY** set a timer at reboot and use the time since reboot, choosing the granularity such that different requests arrive at different times. Servers **MAY** intermittently reset the timer and **MAY** generate a random offset applied to all timestamps. When resetting the timer, the server **MUST** reject all Echo option values that were created before the reset.

Servers that use the "List of Cached Random Values and Timestamps" method described in Appendix A may be vulnerable to resource exhaustion attacks. One way to minimize the state is to use the "Integrity-Protected Timestamp" method described in Appendix A.

5.1. Token Reuse

Reusing Tokens in a way so that responses are guaranteed to not be associated with the wrong request is not trivial. The server may process requests in any order and send multiple responses to the same request. An attacker may block, delay, and reorder messages. The use of a sequence number is therefore recommended when CoAP is used with a security protocol that does not provide bindings between requests and responses, such as DTLS or TLS.

For a generic response to a Confirmable request over DTLS, binding can only be claimed without out-of-band knowledge if:

- the original request was never retransmitted and
- the response was piggybacked in an Acknowledgement message (as a Confirmable or Non-confirmable response may have been transmitted multiple times).

If observation was used, the same holds for the registration, all reregistrations, and the cancellation.

(In addition, for observations, any responses using that Token and a DTLS sequence number earlier than the cancellation Acknowledgement message need to be discarded. This is typically not supported in DTLS implementations.)

In some setups, Tokens can be reused without the above constraints, as a different component in the setup provides the associations:

- In CoAP over TLS, retransmissions are not handled by the CoAP layer and behave like a replay window size of 1. When a client is sending TLS-protected requests without Observe to a single server, the client can reuse a Token as soon as the previous response with that Token has been received.
- Requests whose responses are cryptographically bound to the requests (like in OSCORE) can reuse Tokens indefinitely.

In all other cases, a sequence number approach is **RECOMMENDED**, as per Section 4.

Tokens that cannot be reused need to be handled appropriately. This could be solved by increasing the Token as soon as the currently used Token cannot be reused or by keeping a list of all Tokens unsuitable for reuse.

When the Token (or part of the Token) contains a sequence number, the encoding of the sequence number has to be chosen in a way to avoid any collisions. This is especially true when the Token contains more information than just the sequence number, e.g., the serialized state, as in [RFC8974].

6. Privacy Considerations

Implementations **SHOULD NOT** put any privacy-sensitive information in the Echo or Request-Tag option values. Unencrypted timestamps could reveal information about the server, such as location, time since reboot, or that the server will accept expired certificates. Timestamps **MAY** be used if the Echo option is encrypted between the client and the server, e.g., in the case of DTLS without proxies or when using OSCORE with an Inner Echo option.

Like HTTP cookies, the Echo option could potentially be abused as a tracking mechanism that identifies a client across requests. This is especially true for preemptive Echo option values. Servers **MUST NOT** use the Echo option to correlate requests for other purposes than freshness and reachability. Clients only send Echo option values to the same server from which the values were received. Compared to HTTP, CoAP clients are often authenticated and non-mobile, and servers can therefore often correlate requests based on the security context, the client credentials, or the network address. Especially when the Echo option increases a server's ability to correlate requests, clients **MAY** discard all preemptive Echo option values.

Publicly visible generated identifiers, even when opaque (as all defined in this document are), can leak information as described in [NUMERIC-IDS]. To avoid the effects described there, the absent Request-Tag option should be recycled as much as possible. (That is generally possible as long as a security mechanism is in place -- even in the case of OSCORE outer block-wise transfers, as the OSCORE option's variation ensures that no matchable requests are created by different clients.) When an unprotected Echo option is used to demonstrate reachability, the recommended mechanism of Section 2.3 keeps the effects to a minimum.

7. IANA Considerations

IANA has added the following option numbers to the "CoAP Option Numbers" registry defined by [RFC7252]:

Number	Name	Reference
252	Echo	RFC 9175
292	Request-Tag	RFC 9175

Table 3: Additions to CoAP Option Numbers Registry

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/info/rfc6347>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <https://www.rfc-editor.org/info/rfc7252>.

[RFC7959] Bormann, C. and Z. Shelby, Ed., "Block-Wise Transfers in the Constrained Application Protocol (CoAP)", RFC 7959, DOI 10.17487/RFC7959, August 2016, <https://www.rfc-editor.org/info/rfc7959>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8470] Thomson, M., Nottingham, M., and W. Tarreau, "Using Early Data in HTTP", RFC 8470, DOI 10.17487/RFC8470, September 2018, <https://www.rfc-editor.org/info/rfc8470>.

[RFC8613] Selander, G., Mattsson, J., Palombini, F., and L. Seitz, "Object Security for Constrained RESTful Environments (OSCORE)", RFC 8613, DOI 10.17487/RFC8613, July 2019, <https://www.rfc-editor.org/info/rfc8613>.

8.2. Informative References

[COAP-ATTACKS] Preuß Mattsson, J., Fornehed, J., Selander, G., Palombini, F., and C. Amsüss, "Attacks on the Constrained Application Protocol (CoAP)", Work in Progress, Internet-Draft, draft-mattsson-core-coap-attacks-01, 27 July 2021, <https://datatracker.ietf.org/doc/html/draft-mattsson-core-coap-attacks-01>.

[GROUP-COAP] Dijk, E., Wang, C., and M. Tiloca, "Group Communication for the Constrained Application Protocol (CoAP)", Work in Progress, Internet-Draft, draft-ietf-core-groupcomm-bis-05, 25 October 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-core-groupcomm-bis-05>.

[GROUP-OSCORE] Tiloca, M., Selander, G., Palombini, F., Preuß Mattsson, J., and J. Park, "Group OSCORE - Secure Group Communication for CoAP", Work in Progress, Internet-Draft, draft-ietf-core-oscore-groupcomm-13, 25 October 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-core-oscore-groupcomm-13>.

[NUMERIC-IDS] Gont, F. and I. Arce, "On the Generation of Transient Numeric Identifiers", Work in Progress, Internet-Draft, draft-irtf-pearg-numeric-ids-generation-08, 31 January 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-pearg-numeric-ids-generation-08>.

[REST] Fielding, R., "Architectural Styles and the Design of Network-based Software Architectures", 2000, <https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf>.

[RFC7641] Hartke, K., "Observing Resources in the Constrained Application Protocol (CoAP)", RFC 7641, DOI 10.17487/RFC7641, September 2015, <https://www.rfc-editor.org/info/rfc7641>.

[RFC8323] Bormann, C., Lemay, S., Tschofenig, H., Hartke, K., Silverajan, B., and B. Raymor, Ed., "CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets", RFC 8323, DOI 10.17487/RFC8323, February 2018, <https://www.rfc-editor.org/info/rfc8323>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC8974] Hartke, K. and M. Richardson, "Extended Tokens and Stateless Clients in the Constrained Application Protocol (CoAP)", RFC 8974, DOI 10.17487/RFC8974, January 2021, <https://www.rfc-editor.org/info/rfc8974>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

Appendix A. Methods for Generating Echo Option Values

The content and structure of the Echo option value are implementation specific and determined by the server. Two simple mechanisms for time-based freshness and one for event-based freshness are outlined in this appendix. The "List of Cached Random Values and Timestamps" mechanism is **RECOMMENDED** in general. The "Integrity-Protected Timestamp" mechanism is **RECOMMENDED** in case the Echo option is encrypted between the client and the server.

Different mechanisms have different trade-offs between the size of the Echo option value, the amount of server state, the amount of computation, and the security properties offered. A server **MAY** use different methods and security levels for different use cases (client aliveness, request freshness, state synchronization, network address reachability, etc.).

1. List of Cached Random Values and Timestamps. The Echo option value is a (pseudo)random byte string called r. The server caches a list containing the random byte strings and their initial transmission times. Assuming 72-bit random values and 32-bit timestamps, the size of the Echo option value is 9 bytes and the amount of server state is 13n bytes, where n is the number of active Echo option values. The security against an attacker guessing Echo option values is given by s = bit length of r - log2(n). The length of r and the maximum allowed n should be set so that the security level is harmonized with other parts of the deployment, e.g., s >= 64. If the server loses time continuity, e.g., due to reboot, the entries in the old list **MUST** be deleted.

   Echo option value: random value r

   Server State: random value r, timestamp t0

   This method is suitable for both time-based and event-based freshness (e.g., by clearing the cache when an event occurs) and is independent of the client authority.

2. Integrity-Protected Timestamp. The Echo option value is an integrity-protected timestamp. The timestamp can have a different resolution and range. A 32-bit timestamp can, e.g., give a resolution of 1 second with a range of 136 years. The (pseudo)random secret key is generated by the server and not shared with any other party. The use of truncated HMAC-SHA-256 is **RECOMMENDED**. With a 32-bit timestamp and a 64-bit MAC, the size of the Echo option value is 12 bytes, and the server state is small and constant. The security against an attacker guessing Echo option values is given by the MAC length. If the server loses time continuity, e.g., due to reboot, the old key **MUST** be deleted and replaced by a new random secret key. Note that the privacy considerations in Section 6 may apply to the timestamp. Therefore, it might be important to encrypt it. Depending on the choice of encryption algorithms, this may require an initialization vector to be included in the Echo option value (see below).

   Echo option value: timestamp t0, MAC(k, t0)

   Server State: secret key k

This method is suitable for both time-based and event-based freshness (by the server remembering the time at which the event took place) and independent of the client authority.

If this method is used to additionally obtain network reachability of the client, the server **MUST** use the client's network address too, e.g., as in MAC(k, t0, claimed network address).

3. Persistent Counter. This can be used in OSCORE for sequence number recovery, per Appendix B.1.2 of [RFC8613]. The Echo option value is a simple counter without integrity protection of its own, serialized in uint format. The counter is incremented in a persistent way every time the state that needs to be synchronized is changed (in the case described in Appendix B.1.2 of [RFC8613], when a reboot indicates that volatile state may have been lost). An example of how such a persistent counter can be implemented efficiently is the OSCORE server Sender Sequence Number mechanism described in Appendix B.1.1 of [RFC8613].

   Echo option value: counter

   Server State: counter

   This method is suitable only if the client is the authority over the synchronized property. Consequently, it cannot be used to show client aliveness. It provides statements from the client similar to event-based freshness (but without a proof of freshness).

Other mechanisms complying with the security and privacy considerations may be used. The use of encrypted timestamps in the Echo option provides additional protection but typically requires an initialization vector (a.k.a. nonce) as input to the encryption algorithm, which adds a slight complication to the procedure as well as overhead.

Appendix B. Request-Tag Message Size Impact

In absence of concurrent operations, the Request-Tag mechanism for body integrity (Section 3.5.1) incurs no overhead if no messages are lost (more precisely, in OSCORE, if no operations are aborted due to repeated transmission failure and, in DTLS, if no packets are lost and replay protection is active) or when block-wise request operations happen rarely (in OSCORE, if there is always only one request block-wise operation in the replay window).

In those situations, no message has any Request-Tag option set, and the Request-Tag value can be recycled indefinitely.

When the absence of a Request-Tag option cannot be recycled any more within a security context, the messages with a present but empty Request-Tag option can be used (1 byte overhead), and when that is used up, 256 values from 1-byte options (2 bytes overhead) are available.

In situations where that overhead is unacceptable (e.g., because the payloads are known to be at a fragmentation threshold), the absent Request-Tag value can be made usable again:

- In DTLS, a new session can be established.

- In OSCORE, the sequence number can be artificially increased so that all lost messages are outside of the replay window by the time the first request of the new operation gets processed, and all earlier operations can therefore be regarded as concluded.

Acknowledgements

The authors want to thank Carsten Bormann, Roman Danyliw, Benjamin Kaduk, Murray Kucherawy, Francesca Palombini, and Jim Schaad for providing valuable input to the document.

Authors' Addresses

Christian Amsüss
Email: christian@amsuess.com

John Preuß Mattsson
Ericsson AB
Email: john.mattsson@ericsson.com

Göran Selander
Ericsson AB
Email: goran.selander@ericsson.com