Individual Submission
Internet-Draft
Intended status: Informational
Expires: 12 October 2026

D. Condrey
WritersLogic Inc.

10 April 2026

A Conformant Mechanism for Content Binding in Text Streams
draft-condrey-content-binding-00

Abstract

This document proposes a conformant mechanism for binding metadata to plain text streams using boundary-delimited transport. The mechanism uses existing ASCII characters to establish a text/non-text boundary, requiring no new code points and no changes to existing Unicode text processing algorithms. It defines a delimiter format, a parsing algorithm, a canonicalization procedure, and correctness criteria sufficient for two independent implementations to interoperate without coordination. This RFC is a companion to a Unicode Technical Standard proposal submitted to the Unicode Technical Committee.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 12 October 2026.

Copyright Notice

Table of Contents

1. Introduction

This document specifies a wire format for binding opaque payloads -- signatures, provenance manifests, AI-generated-content markers -- to plain text streams using boundary-delimited transport: a visible ASCII start delimiter, an optional header section, a Base64-encoded payload, and a matching end delimiter. The text preceding the block is preserved byte-for-byte, satisfying what this document calls the *Text Self-Containment Invariant*: text MUST remain semantically complete and self-contained in the absence of any binding mechanism. OpenPGP cleartext signatures [RFC9580] and PEM-encoded cryptographic objects [RFC7468] have used this pattern (Class 2, boundary-delimited transport) for decades. The alternative of hiding metadata inside the text using invisible or repurposed Unicode characters (Class 1,

in-band encoding) is structurally indistinguishable from steganographic attacks such as Trojan Source [TROJAN-SOURCE], GlassWorm [GLASSWORM], and whitespace-replacement techniques [HELLMEIER2025], and has also been flagged for conformance concerns by the Unicode Technical Committee [L2-26-042] [L2-25-241]; Class 1 is rejected throughout this document. A companion Unicode problem statement [L2-26-XXX] presents a related recognition question to the UTC that is orthogonal to this specification.

Editor’s note: L2/26-XXX is a placeholder identifier pending UTC registration and will be updated in a future revision of this draft.

2. Prior Art

Five existing systems occupy nearby points in the design space:

* *OpenPGP cleartext signatures* [RFC9580], 1991. Fixed in-band ASCII delimiter (-----BEGIN PGP SIGNED MESSAGE-----), Base64 payload, visible, survives copy/paste. Direct precedent.
* *PEM / PKIX / CMS* [RFC7468], originally 1993. Fixed in-band ASCII delimiter with payload-type label (-----BEGIN CERTIFICATE-----, etc.), Base64 payload, visible, survives copy/paste. Direct precedent.
* *MIME multipart* [RFC2046], 1996. Boundary token declared out-of-band in a Content-Type header. Does not survive copy/paste because the declaring header is lost.
* *DKIM* [RFC6376], 2007. No body delimiter; signature lives in an SMTP header field. Does not survive copy/paste.
* *Unicode interlinear annotation characters* (U+FFF9-U+FFFB), Unicode 3.0 (1999). Dedicated Cf code points delimit annotated ranges in-band. Invisible to the user, not default-ignorable.

This mechanism follows the PGP/PEM pattern but is payload-agnostic (no label-to-type binding) and defines an explicit header section.

3. Requirements

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

Terms used throughout:

* *Text/non-text boundary*: The point in a text stream at which text processing ends and opaque data begins. Established by a delimiter (Section 4.2). Content before the boundary is text; content after it, up to the corresponding end delimiter, is opaque payload.
* *Content binding block*: A region demarcated by a start delimiter and an end delimiter, containing an optional header section and a Base64-encoded payload.
* *Aware implementation*: Software that recognizes content binding blocks and processes them according to this specification.
* *Unaware implementation*: Software without explicit content binding support. Unaware implementations treat the block as ordinary text.

A conformant mechanism must satisfy:

1. No repurposing of existing Unicode characters outside their defined semantics.
2. The block is logically outside the text for Unicode processing: it must not participate in grapheme cluster determination, bidirectional processing, line breaking, normalization, collation, or default casing. Unaware implementations that treat it as an ordinary paragraph are acceptable.
3. Survives plain-text operations: copy, paste, transfer, and plain-text storage.
4. Unambiguously detectable by aware implementations.
5. Degrades gracefully in unaware implementations: text preserved verbatim, block visible rather than silently stripped.
6. Payload-agnostic via a standard binary encoding (Base64).
7. Clearly distinguishable from adversarial content manipulation.

Payload contents are outside the scope of this document.

4. Specification

4.1. General Structure

A text stream may contain zero or more content binding blocks. The ABNF grammar (Section 4.6.1) defines the structure: text-content *(binding-block [text-content]). A block may appear at the start of the stream, at the end, or between regions of text. Each block consists of a start delimiter, an optional header section, a Base64-encoded payload (Section 4 of [RFC4648]; line-wrapped at 76 characters per Section 6.8 of [RFC2045]), and an end delimiter. Header semantics are defined by higher-level protocols; this document defines only the syntax.

Aware implementations MUST NOT display the raw Base64 payload as if it were text content intended for the user, and SHOULD provide a visual indication that a block is present (e.g., a "content credentials attached" indicator).

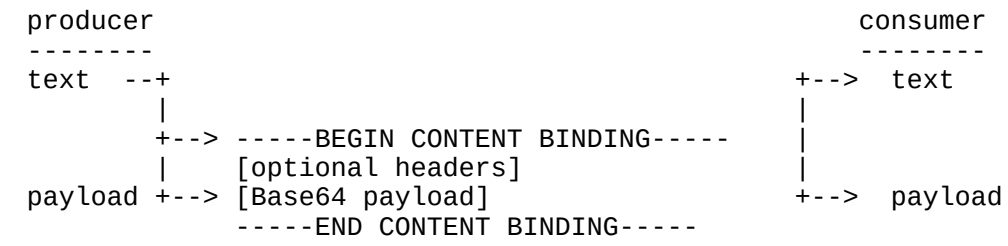

Figure 1: Data flow through a content binding block.

A concrete example with a hypothetical provenance manifest:

The quick brown fox jumps over
the lazy dog.

-----BEGIN CONTENT BINDING-----
Type: application/provenance-manifest+cbor

dGhpcyBpcyBhIHBsYWNlaG9sZGVyIGZvciBh
aWZlc3QgdGhhdCB3b3VsZCBub3JtYWxseSBi
eXRlcyBvZiBCYXNlNjQtZW5jb2RlZCBDQk9S
-----END CONTENT BINDING-----

4.2. Primary Mechanism: ASCII Delimiters (No New Code Points)

The delimiters are -----BEGIN CONTENT BINDING----- and -----END CONTENT BINDING-----.

Delimiter matching MUST be byte-for-byte and case-sensitive. The delimiter MUST occupy an entire line, with no leading or trailing characters other than an optional CR before LF. Visually similar code points (en-dash U+2013, em-dash U+2014, minus sign U+2212) MUST NOT match the ASCII hyphen-minus. Per-line matching needs no lookahead; malformed-block recovery (Section 4.6.2, step 7) rewinds to the recorded block-start, but that is recovery, not detection.

Normative rules:

* Each delimiter MUST appear on its own line. The start delimiter MUST be preceded by a blank line (or appear at the beginning of the text stream). The end delimiter MUST be followed by end-of-text, a blank line, or another start delimiter.

* Line endings within the content binding block (delimiters, headers, and Base64 payload) MUST use LF (U+000A). Implementations MUST accept CRLF and normalize to LF during parsing. The text content preceding the block MAY use any line-ending convention.
* The payload region contains an optional header section followed by Base64-encoded data. Headers, if present, are lines of the form Name: value using only printable ASCII, terminated by a blank line before the Base64 data. The header-name grammar (Section 4.6.1) admits any printable ASCII character except colon; this is deliberately more permissive than MIME tokens, so higher-level protocols with their own naming conventions can use content binding as a transport. Higher-level protocols MAY restrict header names further within their own namespace.
* Implementations MUST support multiple content binding blocks in a single text stream.
* Implementations MUST NOT modify the text content preceding the block.
* Aware implementations MUST NOT present the content binding block as ordinary text content and SHOULD provide a visual indication of its presence.

The delimiter string is theoretically possible in ordinary text, but PGP has shared this risk for three decades without a known collision.

4.3. Requirements Satisfaction

The mechanism satisfies all seven requirements in Section 3:

* _Req 1 (no repurposing)_: ASCII characters are used in their ordinary capacity.
* _Req 2 (no adverse text-processing effect)_: aware implementations exclude the block from grapheme, bidi, line-break, normalization, collation, and casing operations (Unicode Standard Annexes #9, #14, #15, #29 and UTS #10 [UNICODE]); unaware implementations see it as an ordinary ASCII paragraph.
* _Req 3 (survives plain-text operations)_: every character is ASCII and normalization-invariant, as PGP and PEM have demonstrated for decades.
* _Req 4 (unambiguously detectable)_: exact byte-for-byte string match.
* _Req 5 (graceful degradation)_: the block is visible in unaware implementations (like PGP and PEM, unlike DKIM).
* _Req 6 (payload-agnostic)_: opaque Base64; semantics are delegated to higher-level protocols.
* _Req 7 (distinguishable from adversarial manipulation)_: visibility (Section 5.3) combined with exact-match rejection of lookalike delimiters (Section 5.4).

4.4. Normalization

Every character in the delimiters, headers, and Base64 payload is ASCII, and all ASCII code points are invariant under NFC, NFD, NFKC, and NFKD (Unicode Standard Annex #15 [UNICODE]). A content binding block therefore survives any clipboard, storage, or transport layer that applies Unicode normalization. Aware implementations MUST detect and extract the block _before_ applying any normalization or other text transformation to the surrounding text content; otherwise a transformation applied to the stream could alter the delimiters' line context.

4.5. Canonicalization of Text Content

Higher-level protocols that sign or digest text content need a deterministic canonical form; without one, the same logical text produces different byte sequences across systems and verification fails. The canonical form of the text content is:

1. Take the text content preceding the first content binding block, or the entire stream if no block is present.
2. Replace every CR LF (U+000D U+000A) and bare CR (U+000D) with a single LF (U+000A).
3. Preserve any leading UTF-8 BOM (U+FEFF). Higher-level protocols that want to exclude the BOM MUST specify that exclusion themselves.
4. Apply no other modification to the code points. Implementations MUST NOT apply Unicode normalization (NFC, NFD, NFKC, NFKD) as part of canonicalization: applying a normalization at the transport layer would silently alter the text and violate C1. Because some systems silently normalize on clipboard or storage (macOS applies NFD; some databases apply NFC), higher-level protocols that compute signatures SHOULD specify a normalization form (typically NFC) and apply it at both signing and verification time.
5. Encode the resulting code point sequence as UTF-8.

Content binding blocks themselves MUST NOT appear in the canonical form; the signature is computed over the text content only, so including the block would create a circular dependency. If multiple blocks are present, each block's signature covers the same canonical content (the text preceding the first block). Blocks MUST NOT sign each other; higher-level protocols that need chained signatures MUST define their own sequencing rules inside the payload.

4.6. Detection and Parsing

This section gives the formal grammar and normative algorithm for detecting and parsing content binding blocks.

4.6.1. ABNF Grammar

ABNF grammar [RFC5234] for a well-formed content binding block:

```
text-stream       = text-content *(binding-block [text-content])

; text-content: any sequence of Unicode characters not
; containing a start-delimiter at the beginning of a line.
; This production cannot be expressed in ABNF; its boundaries
; are defined by the detection algorithm in Section 4.6.2.

binding-block     = blank-line start-line
                    [header-section] payload-section end-line

start-line        = start-delimiter LF
end-line          = end-delimiter LF / end-delimiter EOF

start-delimiter   = %s"-----BEGIN CONTENT BINDING-----"
end-delimiter     = %s"-----END CONTENT BINDING-----"

header-section    = 1*header-line blank-line
header-line       = header-name ":" SP header-value LF
header-name       = 1*(%x21-39 / %x3B-7E)     ; printable ASCII except ":"
header-value      = *(%x20-7E)                  ; printable ASCII and SP
blank-line        = LF

payload-section   = *base64-line [base64-last]
base64-line       = 1*76base64-char LF
base64-last       = 1*76(base64-char / pad) LF
base64-char       = ALPHA / DIGIT / "+" / "/"
pad               = "="

LF                = %x0A
SP                = %x20
EOF               = ""                          ; end of stream
```

The text-content production cannot be expressed purely in ABNF; its boundaries are defined operationally by the detection algorithm in Section 4.6.2. Implementations MUST accept CR LF (%x0D %x0A) in place of LF in all productions and normalize to LF during parsing.

A leading UTF-8 BOM (U+FEFF) is part of the text content. Implementations MUST NOT strip it before parsing and MUST preserve it in the canonical form (Section 4.5). Some I/O libraries silently strip BOMs on read; applications that use such libraries must re-introduce the BOM or bypass the stripping.

4.6.2. Detection Algorithm

Normative algorithm:

1. Initialize state to SCANNING. Set block-start to null.
2. Read the next line from the text stream. If end-of-stream, go to step 8.
3. If state is SCANNING and the line matches start-delimiter exactly, set state to IN_BLOCK, record block-start position, initialize an empty header list and an empty payload buffer, set sub-state to HEADERS, and go to step 2.
4. If state is IN_BLOCK and sub-state is HEADERS, apply the first matching rule: (4.1) if the line is blank (empty or LF only), set sub-state to PAYLOAD and go to step 2; (4.2) if the line matches header-name ":" SP header-value, append it to the header list and go to step 2; (4.3) otherwise, set sub-state to PAYLOAD and process this line as step 5.
5. If state is IN_BLOCK and sub-state is PAYLOAD, apply the first matching rule: (5.1) if the line matches end-delimiter exactly, go to step 6; (5.2) if the line contains only characters in the Base64 alphabet, padding, and whitespace, append it to the payload buffer and go to step 2; (5.3) otherwise, the block is malformed and go to step 7. The lenient handling of whitespace in 5.2 is consistent with Section 6.8 of [RFC2045]; the ABNF grammar describes the canonical form for conformant producers and does not constrain lenient parsing by consumers.
6. Block complete. Decode the payload buffer as Base64 (Section 4 of [RFC4648]). If decoding fails, the block is malformed; go to step 7. Otherwise, emit a parsed binding block (headers, decoded payload), set state to SCANNING, and go to step 2.
7. Malformed block. Discard accumulated headers and payload buffer. Treat all content from block-start through the current position as ordinary text. Set state to SCANNING. Go to step 2.
8. End of stream. If state is IN_BLOCK, the block is unclosed; treat all content from block-start onward as ordinary text. Emit any accumulated text content. Terminate.

In all cases, the text content preceding and between binding blocks is preserved unmodified.

4.6.3. Error Conditions

Condition	Detection	Required behavior
Unclosed block	End-of-stream reached after start-delimiter without matching end-delimiter	Treat block-start through end-of-stream as ordinary text
Invalid Base64	Characters outside Base64 alphabet, padding, and whitespace in payload region	Reject block; preserve all text verbatim
Truncated Base64	Valid Base64 characters but incorrect padding	Reject block; preserve all text verbatim
Nested start	start-delimiter appears inside an open block's payload region	Treat as malformed payload; reject block
Empty payload	No Base64 lines between headers and end-delimiter	Valid; block carries empty payload
Non-ASCII header	Code points outside U+0020-U+007E in header-name or header-value	Reject block; preserve all text verbatim
Missing blank line	Header lines not terminated by blank line before payload	Parser treats first non-header, non-blank line as payload start (lenient)

Table 1

4.7. Test Vectors

Test vectors for correct parsing and canonicalization. Two implementations that agree on these outputs are interoperable.

Vector 1: Single block, no headers.

Input (LF line endings):

```
Hello, world.
This is a test.

-----BEGIN CONTENT BINDING-----

SGVsbG8=
-----END CONTENT BINDING-----
```

Expected parse result:

* Text content: Hello, world.\nThis is a test. (29 bytes)
* Blocks: 1, no headers, payload = Hello (5 bytes)
* Trailing text: (empty)

Canonical form:

* UTF-8 hex:
  48656c6c6f2c20776f726c642e0a546869732069732061207465737 42e
* SHA-256:
  02b5eda2f3782995430bba0bb2c650fe6f872ae9b253b616da17e81a297c9f43

Vector 2: CRLF normalization.

Same input as Vector 1 but with CR LF (0D 0A) line endings throughout. The canonical form MUST produce the same UTF-8 hex and SHA-256 as Vector 1 after CR LF is normalized to LF.

Vector 3: Malformed block (invalid Base64).

Input:

```
Some text.

-----BEGIN CONTENT BINDING-----

Not valid base64!@#$
-----END CONTENT BINDING-----
```

Expected parse result:

* Text: Some text. (10 bytes)
* Blocks: 0 (rejected as malformed)
* Block region preserved as ordinary text

Malformed blocks MUST NOT be interpreted as valid. No partial decoding or recovery heuristics.

Vector 4: Headers, multiple blocks, interleaved text.

```
Input:

First paragraph.

-----BEGIN CONTENT BINDING-----
Type: application/provenance-manifest+cbor

cHJvdmVuYW5jZSBtYW5pZmVzdCBwbGFj
ZWhvbGRlcg==
-----END CONTENT BINDING-----

Second paragraph.

-----BEGIN CONTENT BINDING-----
Type: application/signature

ZGlnaXRhbCBzaWduYXR1cmUgcGxhY2Vo
b2xkZXI=
-----END CONTENT BINDING-----

Expected parse result:

*  Text: First paragraph. (16 bytes)
*  Block 1: provenance-manifest+cbor, 31 bytes
*  Trailing: Second paragraph.
*  Block 2: signature, 28 bytes

Canonical form of text content (Section 4.5):

*  UTF-8 hex: 4669727374207061726167726170682e
*  SHA-256:
   98ea01bc109a52fdf7145c10c648e8b27b8ebc877aaa79405f20b044ecfcacaa

Two independent reference implementations (Python and Rust) sharing
no code, built against the normative algorithm in Section 4.6.2, are
available at https://github.com/writerslogic/unicode-content-binding.
Both produce identical parse results for the test vectors above,
satisfying C2 and C3.
```

5. Security Considerations

5.1. Security Model

The content binding mechanism provides no authenticity, integrity, or confidentiality guarantees. It defines only a transport for associating opaque data with text content. Authenticity and integrity MUST be established by higher-level protocols operating on the decoded payload. An attacker can construct a syntactically valid block with arbitrary payload content.

5.2. Threat Model

The mechanism operates in an environment where an attacker has full control over the text stream.

Threat	Mit.	By	Notes
T1: Injection	No	HLP	Anyone can construct a valid block. Authentication lives in the payload, not the delimiter.
T2: Removal	No	HLP	Stripping a block is trivial and silent. Protocol must reference expected bindings so absence is detectable.
T3: Reordering	No	HLP	Enforce in payload if ordering matters.
T4: Spoofing	*Yes*	Mech	The one threat this layer closes. Exact byte match; en-dash, em-dash, minus sign all rejected.
T5: Text mod	Partial	Both	C1 preserves text byte-for-byte. Detecting tampering requires the payload signature, not the mechanism.
T6: Payload mod	No	HLP	Same as any Base64 blob: integrity via signatures.
T7: Trailing text	No	HLP+UI	The most subtle attack. Signature covers text before first block only; appended text looks continuous. UI SHOULD mark the boundary.
T8: Replay	No	HLP	Bind signatures to document identity. Without that, any document with identical text validates.

Table 2

5.3. Visibility as a Security Property

The mechanism's defense against confusion with adversarial text manipulation is its visibility (Section 1). A content binding block is visible by default: unaware implementations render it as ordinary text, aware implementations acknowledge it (e.g., collapsed to an indicator). In either case the user can see that metadata is present, and an attacker cannot inject a block without producing a visible artifact. In-band encoding schemes have no analogous property because their entire attack surface is invisible.

5.4. Confusable and Homoglyph Concerns

The ASCII hyphen-minus U+002D in the delimiter has visual lookalikes in Unicode (en-dash U+2013, em-dash U+2014, minus sign U+2212, and others) that an attacker could use to construct a spoofed block. The byte-for-byte matching rule in Section 4.2 closes this avenue: lookalikes are rejected and no lookalike-to-ASCII normalization is performed before comparison.

5.5. Payload Security

Payload security is outside the scope of this document. Base64 encoding confines the payload region to printable ASCII, preventing in-band injection of control characters or bidirectional overrides [UNICODE]. Implementations that decode a payload MUST treat the result as untrusted input.

6. Compatibility

The mechanism does not change the meaning of any existing text. The exact string -----BEGIN CONTENT BINDING----- does not appear in any public code repository (GitHub, as of 2026), any IANA or [RFC7468] label registry, or any known natural-language corpus. PGP, SSH, PEM, and content binding blocks coexist unambiguously because they are distinguished by label, and the "CONTENT BINDING" label is disjoint from the RFC 7468 registry (which covers PEM-type encodings of DER/ ASN.1 structures).

7. Conformance

7.1. Correctness Criteria

An implementation is correct if and only if it satisfies all of these properties:

C1. Text preservation. The text content preceding and between content binding blocks MUST be preserved byte-for-byte, including any leading UTF-8 BOM. Extracting the text content and encoding it as UTF-8 MUST produce a byte sequence identical to the canonical form of the original text content (Section 4.5).

C2. Deterministic boundary detection. Given the same input text stream, all conformant implementations MUST identify the same text/non-text boundaries at the same positions. Boundary detection MUST NOT depend on locale, platform, configuration, or payload content.

C3. Deterministic parsing. Given the same input text stream, all conformant implementations MUST produce the same parse result: the same text content, the same number of blocks, the same headers, and the same decoded payload bytes.

C4. Payload opacity. An implementation MUST NOT interpret, transform, validate, or act on the decoded payload.

C5. All-or-nothing rejection. A malformed block MUST be rejected as a whole. Implementations MUST NOT partially decode the payload, extract a subset of headers, or interpret any portion of a malformed block as valid. On rejection, the entire region from start-delimiter through the point of failure MUST be treated as ordinary text (Section 4.6.3).

C6. Round-trip stability. Parsing a text stream, extracting the text content, and re-appending the same content binding blocks MUST produce a text stream that parses identically to the original.

7.2. Conformance Requirements

A conformant aware implementation MUST satisfy correctness criteria C1 through C6 (Section 7.1), using the delimiter strings from Section 4.2, the detection algorithm from Section 4.6.2, and the error handling from Section 4.6.3.

8. Summary

The mechanism is intended for protocol designers who need to bind signed or opaque data to plain text in environments where out-of-band metadata channels (MIME headers, HTTP headers, file-format wrappers) are unavailable or unreliable. Two independent reference implementations exist at the URL in Section 4.7. Feedback is welcome via the IETF mailing list.

9. IANA Considerations

This document has no IANA actions.

10. References

10.1. Normative References

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996, <https://www.rfc-editor.org/info/rfc2045>.

[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>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[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>.

10.2. Informative References

[GLASSWORM] Dardikman, I., "GlassWorm: First Self-Propagating Worm Using Invisible Code Hits OpenVSX Marketplace", 18 October 2025, <https://www.koi.ai/blog/glassworm-first-self-propagating-worm-using-invisible-code-hits-openvsx-marketplace>.

[HELLMEIER2025] Hellmeier, M., Qarawlus, H., Norkowski, H., and F. Howar, "A Hidden Digital Text Watermarking Method Using Unicode Whitespace Replacement", HICSS 58, 2025.

[L2-25-241] Casper, S., Bommasani, R., Reuel, A., Dai, J., Longpre, S., Bailey, L., and K. Oyin, "UTC Proposal: Watermark

Symbols for AI Training Consent and Text Provenance", Unicode Document Register L2/25-241, October 2025, <https://www.unicode.org/L2/L2025/25241-ai-watermarks.pdf>.

[L2-26-042] Constable, P. and J. Hadley, "Embedded Metadata in 'Plain' Text", Unicode Document Register L2/26-042, 13 January 2026, <https://www.unicode.org/L2/L2026/26042-embedded-metadata-in-plain-text.pdf>.

[L2-26-XXX] Condrey, D., "Text-Processing Exclusion Zones for Boundary-Delimited Regions", Unicode Document Register L2/26-XXX, April 2026.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <https://www.rfc-editor.org/info/rfc2046>.

[RFC6376] Crocker, D., Ed., Hansen, T., Ed., and M. Kucherawy, Ed., "DomainKeys Identified Mail (DKIM) Signatures", STD 76, RFC 6376, DOI 10.17487/RFC6376, September 2011, <https://www.rfc-editor.org/info/rfc6376>.

[RFC7468] Josefsson, S. and S. Leonard, "Textual Encodings of PKIX, PKCS, and CMS Structures", RFC 7468, DOI 10.17487/RFC7468, April 2015, <https://www.rfc-editor.org/info/rfc7468>.

[RFC9580] Wouters, P., Ed., Huigens, D., Winter, J., and Y. Niibe, "OpenPGP", RFC 9580, DOI 10.17487/RFC9580, July 2024, <https://www.rfc-editor.org/info/rfc9580>.

[TROJAN-SOURCE] Boucher, N. and R. Anderson, "Trojan Source: Invisible Vulnerabilities", 32nd USENIX Security Symposium, August 2023, <https://trojansource.codes/>.

[UNICODE] The Unicode Consortium, "The Unicode Standard, Version 17.0.0 -- Core Specification", 9 September 2025, <https://www.unicode.org/versions/Unicode17.0.0/>.

Author's Address

David Condrey
WritersLogic Inc.
Email: david@writerslogic.com