Invisible Code & Hidden Prompts - How Attackers Weaponize Unicode in Repos (and How SAST Can Help) - Cycode

By Shaked Perets

Published: 2025-11-24 · Archived: 2026-04-29 02:12:05 UTC

Introduction: When Code Isn't What It Seems

Imagine reviewing a code commit that looks perfectly ordinary to the naked eye, yet harbors a hidden backdoor. This isn't sci-fi; it's the reality of invisible code attacks. Attackers have learned to embed malicious instructions using Unicode characters that are deceptive to human readers. The compiler sees one thing, while the human reviewer sees another, breaking our traditional security models.

In 2021, researchers revealed "Trojan Source", showing how Unicode could be abused to make harmful code invisible. By 2025, these techniques moved from theory to practice with the **Glassworm** worm, which hid malicious payloads inside Visual Studio Code extensions using invisible bytes.

In this post, we will go deep into the bytes to understand how this works.

Under the Hood: From ASCII to UTF-8

To understand how code becomes invisible, we first need to understand how computers store text.

Decades ago, we relied on **ASCII**. It was a 7-bit system capable of representing 128 characters (A-Z, 0-9, and basic punctuation). In ASCII, "what you see is what you get." There was no room for hidden tricks because every byte had a visible purpose.

Then came **Unicode**. Unicode is a map that assigns a unique number (a "Code Point") to almost every character in every human language, plus symbols and emojis. Finally, we have **UTF-8**, the encoding standard that translates those Unicode numbers into binary bytes that computers can store.

The Rocket Example 🚀

- **To your eye:** You see a rocket emoji: 🚀
- **To Unicode:** It is Code Point U+1F680.
- **To the machine (Hex/Binary):** In UTF-8, this single character requires 4 bytes: F0 9F 9A 80.

Attackers exploit the fact that Unicode contains thousands of valid code points that map to *nothing* visually, yet still occupy bytes in the file.

Unicode is the universal character encoding standard that lets software handle text in all languages and scripts. Within Unicode's enormous range are numerous non-printing or formatting characters – things like zero-width spaces, directional markers, and private-use symbols. They generally have no visible representation in editors.

Developers “read what they see” in code, but invisible Unicode means we can’t always trust our eyes: parts of the code could be there, doing something nefarious, yet not visually apparent.

Attackers weaponize these properties to hide logic, alter program flow, or conceal malware in source code. In effect, it’s an attack on the human reviewers and tooling: the code compiles and runs normally, but critical pieces are camouflaged from reviewers or basic text diffs.

There are a few main categories of Unicode trickery we need to understand:

- **Variation Selectors** – exploiting characters meant to modify preceding symbols (like emojis) by placing them in isolation, creating invisible sequences that are ignored by editors but processed by the compiler.
- **Private Use Area (PUA) encoding** – using custom, non-standard Unicode characters that render as “nothing” to encode malicious code.
- **Bidirectional (bidi) control characters** – injecting Unicode control codes that reorder text display (intended for mixing RTL and LTR scripts) to visually reshape code, e.g. making code appear as a comment or string when it’s actually executable.

Let’s examine the most dangerous techniques in depth, looking at the raw hex data to see the truth.

Technique 1: Variation Selectors (The Glassworm Method)

Variation Selectors are intended to slightly modify the character preceding them (like changing an emoji color). However, attackers use them in isolation or chains. Because there is no base character to modify, editors simply display nothing – but the bytes remain.

To see why this is dangerous, let’s compare a truly empty string against a malicious one.

Case A: The Normal “Empty” String

This is what safe code looks like. A developer initializes a variable with an empty string.

```
const key = "";
```

The Hexdump: Everything is exactly as it seems. The file is small, and the bytes match the text 1:1.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

00000000 63 6f 6e 73 74 20 6b 65 79 20 3d 20 22 22 3b |const key = "";|

00000000 63 6f 6e 73 74 20 6b 65 79 20 3d 20 22 22 3b |const key = "";|

```
00000000 63 6f 6e 73 74 20 6b 65 79 20 3d 20 22 22 3b |const key = "";|
```

Case B: The Weaponized “Empty” String

This is what the **Glassworm** attack looks like. Visually, the code looks *identical* to Case A. const key = “”; (Looks empty, right?)

The Hexdump: Look at the highlighted bytes. Between the quote marks (22), there are now 3 extra bytes (ef b8 80). These represent **Variation Selector-1**, which is invisible. In real attacks, this “empty” space can contain thousands of bytes of malicious payload.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

00000000 63 6f 6e 73 74 20 6b 65 79 20 3d 20 22 ef b8 80 |const key = "...|

00000010 22 3b |";|

00000000 63 6f 6e 73 74 20 6b 65 79 20 3d 20 22 ef b8 80 |const key = "...| 00000010 22 3b |";|

```
00000000  63 6f 6e 73 74 20 6b 65  79 20 3d 20 22 ef b8 80  |const key = "...|
00000010  22 3b                                             |";|
```

Anatomy of the Attack:

- 🟢 **Safe Prefix:** const key = “
  - Normal ASCII bytes ending at 22.
- 🔴 **The Trap:** ef b8 80
  - **This is the invisible code.** These 3 bytes represent U+FE00 (Variation Selector-1). They render as nothing, but the compiler reads them as valid data.
- 🟢 **Safe Suffix:** “;
  - The closing quote 22 and semicolon 3b.

References:

- **Variation Selector-1 (VS1):** https://unicode-explorer.com/b/FE00
- **Variation Selector-17 (VS17):** https://unicode-explorer.com/b/E0100

Technique 2: Private Use Area (PUA) Encoding

Unicode reserves blocks of code points called “Private Use Areas.” These are strictly for internal use by applications and have no standard glyphs. Because no font has a picture for them, editors render them as nothing or a generic “missing character” box which is easily overlooked.

The Attack: Attackers map their malicious script to these characters. For example, they might shift the standard letter ‘A’ (0x41) to a PUA character (U+E0041). The result is a “blank” blob of text that the program decodes at runtime into executable code.

Case A: The Normal String

Here is a standard variable assignment. The variable payload is truly empty.

```
var payload = "";
```

The Hexdump: The hex bytes perfectly match the visible text.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

00000000 76 61 72 20 70 61 79 6c 6f 61 64 20 3d 20 22 22 |var payload = ""|

00000010 3b |;|

00000000 76 61 72 20 70 61 79 6c 6f 61 64 20 3d 20 22 22 |var payload = ""| 00000010 3b |;|

```
00000000  76 61 72 20 70 61 79 6c  6f 61 64 20 3d 20 22 22  |var payload = ""|
00000010  3b                                                |;|
```

Case B: The Weaponized PUA String

Here, the attacker hides a malicious instruction inside what looks like an empty string.

`var payload = " ";` (To the eye, this looks identical to Case A: “”)

The Hexdump: The file size has grown, and new bytes have appeared. The ASCII column shows dots …. Because the terminal cannot display the character, but the Hex column reveals the truth.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

00000000 76 61 72 20 70 61 79 6c 6f 61 64 20 3d 20 22 f3 |var payload = ".|

00000010 bf be 80 22 3b

00000000 76 61 72 20 70 61 79 6c 6f 61 64 20 3d 20 22 f3 |var payload = ".| 00000010 bf be 80 22 3b

```
00000000  76 61 72 20 70 61 79 6c  6f 61 64 20 3d 20 22 f3  |var payload = ".|
00000010  bf be 80 22 3b
```

Anatomy of the Attack:

- 🟢 **Safe Prefix:** var payload = “
  - Standard text.
- 🔴 **The Trap:** f3 bf be 80
  - **Hidden Payload.** This is the UTF-8 representation of the PUA character U+FFF80. It occupies **4 bytes** of disk space but renders **0 pixels** of width.
- 🟢 **Safe Suffix:** “;
  - Closing syntax.

References:

- **Private Use Area Search:** https://unicode-explorer.com/search/
- **Supplementary PUA-A (U+FFF80):** https://unicode-explorer.com/b/FFF80

Supplementary PUA-B (U+10FF80): https://unicode-explorer.com/b/10FF80

Technique 3: Bidirectional Control (Trojan Source)

This technique was famously documented in the research paper Trojan Source. It exploits Unicode characters intended for Right-to-Left languages (like Hebrew or Arabic) to decouple the **visual order** of code from the **logical order** that the compiler executes.

The Attack (The “Stretched String”): In this specific example (adapted from Nick Boucher’s Trojan Source Repo), an attacker uses the **Right-to-Left Override (RLO)** to trick the code reviewer into thinking a string comparison is safe.

- **The Goal:** Force the code to print “You are an admin” even though the user is not an admin.
- **The Method:** The attacker hides the comment // Check if admin *inside* the string literal using invisible Bidi control characters.

What the Reviewer Sees (Visual)

To a human eye, this looks like a standard security check. The code compares accessLevel to “user”. Since they are equal, the if block should *not* run.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

var accessLevel = "user";

if (accessLevel != "user") { // Check if admin

console.log("You are an admin.");

}

var accessLevel = "user"; if (accessLevel != "user") { console.log("You are an admin."); } // Check if admin

```
var accessLevel = "user";
if (accessLevel != "user") { // Check if admin
    console.log("You are an admin.");
}
```

Wait… that comment looks strange? In a vulnerable editor, the comment // Check if admin would appear *outside* the quote marks, making it look like: if (accessLevel != “user”) { // Check if admin

What the Compiler Sees (Logical)

The compiler reads the bytes linearly. It seems that the “comment” is actually part of the string. **Logic:** if (“user” != “user [invisible_bytes] // Check if admin”)

Since “user” is NOT equal to that long garbage string, the condition is **TRUE**, and the code grants admin privileges.

The Evidence (Hexdump): The hexdump -C exposes the trick. Look at how the Bidi characters (e2 80 ae) flip the rendering.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

00000000 69 66 20 28 61 63 63 65 73 73 4c 65 76 65 6c 20 |if (accessLevel |

00000010 21 3d 20 22 75 73 65 72 e2 80 ae 20 e2 81 a6 2f |!= "user... .../|

00000020 2f 20 43 68 65 63 6b 20 69 66 20 61 64 6d 69 6e |/ Check if admin|

00000030 e2 81 a9 20 e2 81 a6 22 29 20 7b 0a |... ...") {.|

00000000 69 66 20 28 61 63 63 65 73 73 4c 65 76 65 6c 20 |if (accessLevel | 00000010 21 3d 20 22 75 73 65 72 e2 80 ae 20 e2 81 a6 2f |!= "user... .../| 00000020 2f 20 43 68 65 63 6b 20 69 66 20 61 64 6d 69 6e |/ Check if admin| 00000030 e2 81 a9 20 e2 81 a6 22 29 20 7b 0a |... ...") {.|

```
00000000  69 66 20 28 61 63 63 65  73 73 4c 65 76 65 6c 20  |if (accessLevel |
00000010  21 3d 20 22 75 73 65 72  e2 80 ae 20 e2 81 a6 2f  |!= "user... .../|
00000020  2f 20 43 68 65 63 6b 20  69 66 20 61 64 6d 69 6e  |/ Check if admin|
00000030  e2 81 a9 20 e2 81 a6 22  29 20 7b 0a              |... ...") {.|
```

Anatomy of the Attack:

- 🟢 **Safe Start:** if (accessLevel != “user
  - Normal code up to byte 72.
- 🔴 **The Trigger:** e2 80 ae
  - **Right-to-Left Override (RLO).** This byte sequence forces the editor to display the *next* characters backwards.
- 🔴 **The Trick:** 2f 2f (//)
  - The RLO makes these slashes appear at the *end* of the line as a comment. In reality (and in the hex), they are trapped **inside** the string literal.

The Fix: GitHub’s Warning

Because this attack is so subtle, platforms like GitHub had to intervene. If you view a file with these hidden characters on GitHub today, you will see a big yellow warning banner:

“This file contains bidirectional Unicode text that may be interpreted or compiled differently than what appears below.”

References:

- **Trojan Source Research:** https://trojansource.codes/
- **JavaScript Proof-of-Concept:** https://github.com/nickboucher/trojan-source/tree/main/JavaScript

GitHub Warning Announcement: https://github.blog/changelog/2021-10-31-warning-about-bidirectional-unicode-text/

The AI Blind Spot: Invisible Prompt Injection

As software development shifts toward AI-driven workflows, a new and dangerous vector has emerged. Developers act as the “eyes” checking the code, but AI agents act as the “brain” reading the raw bytes.

The Technique: Unicode Tag Characters Attackers use **Unicode Tag Characters** (Block U+E0000). These are special versions of standard letters (like A, B, C) that are defined by Unicode to be **invisible** for display. They occupy 4 bytes of storage each but render 0 pixels on the screen.

The Attack Scenario: An attacker wants to trick an AI agent into ignoring safety rules. They inject these invisible tags into a markdown file (like agents.md).

1. **The Human View (Visual)** When a developer reviews the file, they see only the safe text. The invisible characters are rendered as zero-width, effectively disappearing.

```
System: You are a helpful assistant.
```

2. **The AI View (Logical)** The AI doesn't "look" at the screen; it processes the text stream. When it reads the file, it receives the standard text **plus** the invisible tag characters. If the tokenizer processes these tags, the AI reads:

```
System: You are a helpful assistant. IGNORE
```

The Evidence (Hexdump) Here is a hexdump of that exact string. The text appears to end at the dot (.), but the binary data continues.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

00000000 73 73 69 73 74 61 6e 74 2e 20 f3 a0 81 89 f3 a0 |ssistant. ......|

00000010 81 87 f3 a0 81 8e f3 a0 81 8f f3 a0 81 92 f3 a0 |................|

00000020 81 85

00000000 73 73 69 73 74 61 6e 74 2e 20 f3 a0 81 89 f3 a0 |ssistant. ......| 00000010 81 87 f3 a0 81 8e f3 a0 81 8f f3 a0 81 92 f3 a0 |................| 00000020 81 85

```
00000000  73 73 69 73 74 61 6e 74  2e 20 f3 a0 81 89 f3 a0  |ssistant. ......|
00000010  81 87 f3 a0 81 8e f3 a0  81 8f f3 a0 81 92 f3 a0  |................|
00000020  81 85
```

Decoding the Invisible Bytes: The dots (.) in the right-hand column hide the truth, but the hex bytes on the left tell the story.

- **f3 a0 81 89**: This is the invisible **Tag 'I'** (U+E0049). It is completely different from the visible 'I' (0x49), but the AI can read it.
- **f3 a0 81 87**: This is the invisible **Tag 'G'** (U+E0047).

- **f3 a0 81 8e**: This is the invisible **Tag ‘N’** (U+E004E).

The Danger: Because the invisible text is technically part of the prompt, a malicious instruction can be smuggled past human review. The developer approves the “safe” prompt, but the AI executes the hidden “unsafe” command.

Defending Against Invisible Attacks with SAST

Defending against something you literally cannot see is a challenge for humans. However, as we proved with the hexdumps above, these attacks leave a clear trace in the binary data.

Static Application Security Testing (SAST) tools, like Cycode, don’t look at the rendered font – they look at the hex.

1. **Pattern Matching:** SAST can ban specific byte sequences (like EF B8 80 / VS1) that should never appear in source code.
2. **Anomaly Detection:** A SAST scan can flag strings that contain a high density of PUA characters or mixed-script Homoglyphs.

Quick Manual Checks: If you suspect a snippet of code contains invisible characters but don’t have a scanner handy, you can use online detectors to reveal them:

- Invisible Character Detector
- Invisible Character Viewer

By integrating SAST into your CI/CD pipeline, you ensure that no matter how “invisible” the code looks to your eyes, the raw bytes are analyzed, flagged, and blocked before they can hurt your users.

Originally published: November 24, 2025
