GLASSWORM

Variation Selector PE Encoding

How a Full PE Binary Hides Inside an Invisible Unicode String

March 2026 | TLP:WHITE

1. Overview

The GLASSWORM campaign employs a technique not previously documented in public threat intelligence: the encoding of a complete Windows PE binary (DLL or EXE) as a sequence of invisible Unicode Variation Selector characters embedded directly in JavaScript source code. The encoded payload is indistinguishable from an empty string to any human reviewer or static analysis tool that does not specifically scan for high-density Variation Selector codepoints.

This document describes the encoding scheme in precise technical detail, provides worked examples, quantifies detection opportunity at each layer of the attack, and specifies YARA rules and scanner logic sufficient to detect the technique in npm packages, GitHub repositories, and endpoint JavaScript files.

A companion reference implementation (vs_codec.py) provides a working Python encoder and decoder for use in detection rule validation and incident response.

2. Background: Unicode Variation Selectors

Unicode defines Variation Selectors as codepoints whose sole defined purpose is to request an alternate glyph for the immediately preceding character. They have no independent visual representation — by definition, they render as nothing. Two blocks are relevant to this technique:

Block	Codepoint Range	Count	Notes
Variation Selectors	U+FE00 – U+FE0F	16	Used by GLASSWORM. 4 bits of data capacity per codepoint.
Variation Selectors Supplement	U+E0100 – U+E01EF	240	Extended block. Not observed in GLASSWORM samples.

The basic block (U+FE00–U+FE0F) contains exactly 16 codepoints — one for each possible 4-bit nibble value. This is the property that GLASSWORM exploits: each Variation Selector in this block can carry exactly one nibble of payload data. Two codepoints encode one byte.

Crucially, all 16 codepoints in this block are valid Unicode, accepted by every conformant UTF-8 encoder, and rendered as zero visible characters in every major editor, terminal, browser, code review tool, and diff viewer. They pass through git commits, npm publishes, and code review platforms without any visible trace.

3. The Encoding Scheme

3.1 Byte-to-Codepoint Mapping

The encoding is straightforward. For each byte of the PE payload:

- Split the byte into its high nibble (bits 7–4) and low nibble (bits 3–0).
- Map the high nibble to codepoint U+FE00 + high_nibble.
- Map the low nibble to codepoint U+FE00 + low_nibble.
- Emit the two codepoints in order (high nibble first).

This produces exactly 2 Variation Selector codepoints per byte of payload. The mapping is bijective — every possible byte value maps to a unique pair of codepoints, and the scheme is trivially reversible.

3.2 Worked Example: The MZ Header

A Windows PE file always begins with the two-byte magic number 0x4D 0x5A ('MZ'). Encoding these two bytes:

Byte	Hex	High Nibble	Low Nibble	Codepoint 1	Codepoint 2
M	0x4D	0x4 (4)	0xD (13)	U+FE04	U+FE0D
Z	0x5A	0x5 (5)	0xA (10)	U+FE05	U+FE0A

The four codepoints U+FE04 U+FE0D U+FE05 U+FE0A are all invisible. Placed inside a JavaScript backtick string, they appear as:

```
const payload = ``;   // appears empty — actually contains 4 invisible codepoints
```

A hex dump of the UTF-8 encoding of these four codepoints shows:

```
EF B8 84   EF B8 8D   EF B8 85   EF B8 8A
```

Each Variation Selector in the basic block encodes as a 3-byte UTF-8 sequence in the range EF B8 80 – EF B8 8F. This range is the UTF-8 representation of U+FE00–U+FE0F and is itself a reliable detection signature at the byte level.

3.3 Full Encoding Pipeline

The complete pipeline from PE file to JavaScript source is:

- Read the PE binary as raw bytes.
- For each byte, emit two Variation Selector codepoints as described above.
- Concatenate all codepoints into a single string.
- Embed the string inside a JavaScript backtick literal (or any string delimiter).
- Optionally insert an anchor character (e.g. a space or zero-width no-break space U+FEFF) before the first codepoint — the decoder skips non-VS characters, so anchors are cosmetic.

3.4 Capacity and Size Overhead

The encoding introduces a fixed size expansion factor:

Metric	Value	Notes
Codepoints per payload byte	2	One per nibble
UTF-8 bytes per codepoint	3	All U+FE00–U+FE0F are 3-byte sequences
UTF-8 bytes per payload byte	6	Net expansion: 6×
100 KB PE → UTF-8 source size	~600 KB	Within normal JS file size range
1 MB PE → UTF-8 source size	~6 MB	Unusually large but not impossible

For typical second-stage DLLs in the 50–200 KB range, the resulting source file is 300 KB – 1.2 MB. This is larger than a typical npm utility package but not large enough to trigger automatic rejection by npm or most CI pipelines without specific size checks.

4. The JavaScript Layer

4.1 Decoder Structure

The JavaScript-side decoder observed in GLASSWORM-adjacent campaigns follows this pattern:

```
function s(str) {
  const nibbles = [];
  for (const ch of str) {
    const cp = ch.codePointAt(0);
    if (cp >= 0xFE00 && cp <= 0xFE0F) nibbles.push(cp - 0xFE00);
  }
  const bytes = new Uint8Array(nibbles.length / 2);
  for (let i = 0; i < bytes.length; i++)
    bytes[i] = (nibbles[i*2] << 4) | nibbles[i*2 + 1];
  return bytes;
}

eval(Buffer.from(s(`□□□...`)).toString('utf-8'));
```

The function s() iterates the string character by character using codePointAt(), filters for VS codepoints, extracts their nibble values, and reconstructs the byte array. The eval() call then executes the decoded content as JavaScript — or, in the GLASSWORM variant, passes the decoded bytes to the native Bootstrap DLL via the NAPI bridge rather than evaluating them as JS.

4.2 The GLASSWORM Variant: Native Execution Without eval()

The GLASSWORM Bootstrap DLL eliminates eval() entirely by moving the decoding into native code. The JavaScript layer is reduced to:

```
const addon = require('./build/Release/addon.node');
addon.run(`□□□...`);
```

The string containing the invisible payload is passed directly to the Bootstrap DLL via napi_get_value_string_utf16, which receives it as a UTF-16 string. The DLL then calls

WideCharToMultiByte to convert to UTF-8, walks the byte sequence to extract VS codepoints, reconstructs the payload bytes, and uses NtAllocateVirtualMemory and NtProtectVirtualMemory (resolved directly from ntdll.dll to bypass EDR hooks) to load and execute the payload as a PE file in memory via the memexec-0.2.0 Rust crate.

The absence of eval() means JavaScript-layer eval() detection rules produce zero signal. The payload never appears in any form that Node.js's script engine sees as code. The only observable events are the NAPI function call and the subsequent NT syscalls — both of which are invisible to Win32-layer monitoring.

5. In-Memory PE Execution via memexec

Once the Bootstrap has decoded the Variation Selector string into a byte buffer containing a valid PE file, it passes the buffer to the memexec-0.2.0 Rust crate for execution. The memexec crate implements a complete PE loader in Rust:

- Validates the MZ and NT headers (peparser/pe.rs, peparser/header.rs).
- Maps PE sections into memory with correct alignment (peparser/section.rs).
- Applies base relocations if the image could not be loaded at its preferred base address.
- Resolves imports against the host process's loaded modules.
- Calls the PE entry point (DllMain for DLLs, the entry point for EXEs).

This means the second-stage payload — which in GLASSWORM is the C2 agent, Chrome credential harvester, or C/C++ credential stealer — is never written to disk in any form. It exists only as invisible characters in a JavaScript source file and as a memory-mapped region in the Node.js process at runtime.

The combination of Variation Selector encoding, native DLL decoding, direct NT syscalls, and memexec-based loading produces a payload delivery chain with zero disk artefacts and no Win32 API surface visible to standard endpoint monitoring.

6. Detection

Despite its sophistication, this technique is detectable at multiple layers. Detection rules at the source layer are the most reliable because the encoding is structural and cannot be obfuscated further without abandoning the invisibility property.

6.1 Source-Level Detection: UTF-8 Byte Pattern

Every Variation Selector in the U+FE00–U+FE0F block encodes to the UTF-8 byte sequence EF B8 8x (where x is 0–F). A long contiguous run of these sequences in a JavaScript file is anomalous with overwhelming probability. Legitimate JavaScript has no reason to contain more than a handful of Variation Selectors.

YARA rule — detects any JS file with a Variation Selector run long enough to encode at least a minimal PE stub (~64 bytes = 128 codepoints = 384 UTF-8 bytes):

```
rule GLASSWORM_VS_Encoding_JS {
  meta:
    description = "Variation Selector PE encoding in JavaScript source"
    reference   = "GLASSWORM campaign — vs_encoding_technical.docx"
    date        = "2026-03"  tlp = "WHITE"
  strings:
    // 8 consecutive VS codepoints (16 bytes payload) — low-noise threshold
    $vs_run_16 = { EF B8 8? EF B8 8? EF B8 8? EF B8 8?
                   EF B8 8? EF B8 8? EF B8 8? EF B8 8? }
    // MZ magic encoded as VS codepoints: 0x4D 0x5A
    // High/low nibbles: 4,D,5,A → U+FE04 U+FE0D U+FE05 U+FE0A
    $vs_mz = { EF B8 84 EF B8 8D EF B8 85 EF B8 8A }
  condition:
    $vs_run_16 or $vs_mz
}
```

6.2 Source-Level Detection: High-Density VS Scan (Python)

The companion vs_codec.py script includes an analyse_js_file() function that scans a JS file for contiguous Variation Selector runs, reports their lengths, and checks each run for MZ magic. This is suitable for integration into npm package scanning pipelines:

```
from vs_codec import analyse_js_file

result = analyse_js_file('suspicious_package/index.js')
if result['max_run'] > 32 or result['decoded_mz']:
    print('ALERT: possible VS-encoded payload')
    print(f"  longest run : {result['max_run']} codepoints")
    print(f"  decoded MZ  : {result['decoded_mz']}")
```

6.3 Binary-Level Detection: Bootstrap DLL

The Bootstrap DLL that performs the native decode is detectable independently of the JavaScript layer:

```
rule GLASSWORM_Bootstrap_VS_Decoder {
  meta:
    description = "GLASSWORM Bootstrap — VS decoder + NT syscall + memexec"
    date        = "2026-03"  tlp = "WHITE"
  strings:
    $napi     = "napi_register_module_v1" ascii
    $nt_alloc = "NtAllocateVirtualMemory" ascii
    $nt_prot  = "NtProtectVirtualMemory" ascii
    $ntdll    = "ntdll.dll" ascii
    $memexec  = "memexec-0.2.0" ascii
    $neon     = "neon-1.0.0" ascii
    $err      = "NtAllocVmErrNtProtectVmErrInvalidUtf8String" ascii
  condition:
    uint16(0) == 0x5A4D and
    $napi and $ntdll and $nt_alloc and $nt_prot and
    ($memexec or $neon or $err)
}
```

6.4 Behavioural Detection

At runtime, the following behavioural indicators are observable without source access:

- NtAllocateVirtualMemory called from a process whose image path contains node_modules — normal Node.js execution does not call NT memory APIs directly.
- NtProtectVirtualMemory called on a region immediately after NtAllocateVirtualMemory in the same process, changing protection to PAGE_EXECUTE_READ.
- A NAPI function invocation where the JavaScript argument is a string with codepoints exclusively in the U+FE00–U+FE0F range — detectable by Node.js instrumentation hooking napi_get_value_string_utf16.
- Memory-mapped PE image appearing in the Node.js process address space with no corresponding file on disk (detectable via memory forensics tools such as Volatility or pe-sieve).

6.5 Property-Based Defense: Unicode Normalization Enforcement

The most durable defense is not signature-based but property-based: enforce that all JavaScript source files in a codebase or package pass Unicode NFKC normalization without change. NFKC normalization strips Variation Selectors entirely (they have no canonical decomposition that survives normalization). Any file containing VS codepoints will differ from its normalized form, making the check trivially automatable as a CI gate:

```
import unicodedata, sys

def check_file(path):
    text = open(path, encoding='utf-8').read()
    if unicodedata.normalize('NFKC', text) != text:
        print(f'FAIL: {path} contains non-NFKC characters')
        sys.exit(1)
```

This check requires no knowledge of the specific encoding scheme. It catches Variation Selectors, zero-width characters, homoglyphs, and other invisible or visually ambiguous Unicode regardless of how they are used. A package that fails NFKC normalization should be treated as suspicious by default.

7. Indicators

7.1 Encoding Signatures

Indicator	Type	Notes
EF B8 8x repeated (x in 0–F)	UTF-8 byte pattern	VS codepoint sequence in source; run length > 32 bytes is anomalous
EF B8 84 EF B8 8D EF B8 85 EF B8 8A	UTF-8 byte pattern	Encoded MZ magic (0x4D 0x5A) — high confidence PE encoding
U+FE00–U+FE0F in string literals	Unicode property	Any occurrence in require() paths or function arguments is suspicious

NFKC normalization mismatch	Unicode property	Reliable property-based detection regardless of encoding variant

7.2 Bootstrap DLL Strings

String	Source	Significance
ntdll.dllNtAllocateVirtualMemoryNtProtectVirtualMemory	Both samples	Runtime API name strings used for direct NT syscall resolution
NtAllocVmErrNtProtectVmErrInvalidUtf8String	Both samples	Packed Rust error enum — confirms VS decode failure path
memexec-0.2.0	Both samples	Rust crate for in-memory PE execution — confirms payload type
neon-1.0.0	Both samples	Rust NAPI bridge — confirms Node.js native addon delivery

8. Relationship to GLASSWORM Threat Intelligence Report

This document is a technical supplement to the GLASSWORM Threat Intelligence Report (v4). It covers the JavaScript-layer encoding technique in depth; the main report covers the binary components (Neon Bootstrap, Chrome Harvester, C2 Agent, C/C++ Credential Stealer), YARA rules for the binary layer, and full IOC tables.

The key finding that connects the two documents is the memexec-0.2.0 Rust crate identified in both Bootstrap samples. Its presence confirms that what the Bootstrap decodes from the Variation Selector string is a complete PE file, not raw shellcode or an interpreted bytecode format. The JavaScript layer delivers the PE invisibly; the Bootstrap decodes and executes it entirely in memory.

Together, the two documents provide complete coverage of the GLASSWORM delivery chain from npm package publication through to in-memory PE execution.

Appendix A: Encoding Reference

Complete nibble-to-codepoint mapping for the basic Variation Selector block:

Nibble (hex)	Nibble (dec)	Codepoint	UTF-8 bytes

0x0	0	U+FE00	EF B8 80
0x1	1	U+FE01	EF B8 81
0x2	2	U+FE02	EF B8 82
0x3	3	U+FE03	EF B8 83
0x4	4	U+FE04	EF B8 84
0x5	5	U+FE05	EF B8 85
0x6	6	U+FE06	EF B8 86
0x7	7	U+FE07	EF B8 87
0x8	8	U+FE08	EF B8 88
0x9	9	U+FE09	EF B8 89
0xA	10	U+FE0A	EF B8 8A
0xB	11	U+FE0B	EF B8 8B
0xC	12	U+FE0C	EF B8 8C
0xD	13	U+FE0D	EF B8 8D
0xE	14	U+FE0E	EF B8 8E
0xF	15	U+FE0F	EF B8 8F

Appendix B: vs_codec.py Usage Reference

The companion Python script vs_codec.py provides a reference encoder and decoder for use in detection validation and incident response. All commands operate on files:

Command	Usage	Description
encode	vs_codec.py encode <binary> <output.js>	Encode a PE/binary file to a JS snippet containing the invisible payload
decode	vs_codec.py decode <input.js> <output_binary>	Extract and decode VS payload from a JS file back to raw bytes
test	vs_codec.py test <binary>	Round-trip encode then decode and verify byte-for-byte match
selftest	vs_codec.py selftest	Run built-in correctness tests (no files required)

The analyse_js_file() function in vs_codec.py is intended for integration into npm package scanning pipelines. It returns VS run statistics and flags any run that decodes to a PE magic number (MZ).