Commit 4398b78

serhiy-storchaka authored on May 26, 2025 · 24 / 26 · Verified

```
[3.12] gh-133767: Fix use-after-free in the unicode-escape decoder with an error
handler (GH-129648) (GH-133944) (#134337)

If the error handler is used, a new bytes object is created to set as
the object attribute of UnicodeDecodeError, and that bytes object then
replaces the original data. A pointer to the decoded data will became invalid
after destroying that temporary bytes object. So we need other way to return
the first invalid escape from _PyUnicode_DecodeUnicodeEscapeInternal().

_PyBytes_DecodeEscape() does not have such issue, because it does not
use the error handlers registry, but it should be changed for compatibility
with _PyUnicode_DecodeUnicodeEscapeInternal().
(cherry picked from commit 9f69a58)
(cherry picked from commit 6279eb8)
```

3.12 (#134337) · v3.12.13 ••• v3.12.11

1 parent 310cd89 commit 4398b78

8 files changed +194 -57

Top

Filter files…

- Include/cpython
  - bytesobject.h
  - unicodeobject.h
- Lib/test
  - test_codeccallbacks.py
  - test_codecs.py
- Misc/NEWS.d/next/Security
  - 2025-05-09-20-22-54.gh-issue-133767.kN2i3Q.rst

- Objects
  - bytesobject.c
  - unicodeobject.c
- Parser
  - string_parser.c

Search within code

Include/cpython/bytesobject.h

```
@@ -25,6 +25,10 @@ PyAPI_FUNC(PyObject*) _PyBytes_FromHex(
     int use_bytearray);

 /* Helper for PyBytes_DecodeEscape that detects invalid escape chars. */
+PyAPI_FUNC(PyObject*) _PyBytes_DecodeEscape2(const char *, Py_ssize_t,
+                                             const char *,
+                                             int *, const char **);
+// Export for binary compatibility.
 PyAPI_FUNC(PyObject *) _PyBytes_DecodeEscape(const char *, Py_ssize_t,
                                              const char *, const char **);
```

Include/cpython/unicodeobject.h

```
@@ -684,6 +684,19 @@ PyAPI_FUNC(PyObject*)
 _PyUnicode_DecodeUnicodeEscapeStateful(
 );
 /* Helper for PyUnicode_DecodeUnicodeEscape that detects invalid escape
    chars. */
+PyAPI_FUNC(PyObject*) _PyUnicode_DecodeUnicodeEscapeInternal2(
+     const char *string,     /* Unicode-Escape encoded string */
+     Py_ssize_t length,      /* size of string */
+     const char *errors,     /* error handling */
+     Py_ssize_t *consumed,   /* bytes consumed */
+     int *first_invalid_escape_char, /* on return, if not -1, contain the first
+                                        invalid escaped char (<= 0xff) or
 invalid
+                                        octal escape (> 0xff) in string. */
```

```
695  +     const char **first_invalid_escape_ptr); /* on return, if not NULL, may
696  +                                             point to the first invalid escaped
697  +                                             char in string.
698  +                                             May be NULL if errors is not NULL. */
699  + // Export for binary compatibility.
687  700   PyAPI_FUNC(PyObject*) _PyUnicode_DecodeUnicodeEscapeInternal(
688  701           const char *string,     /* Unicode-Escape encoded string */
689  702           Py_ssize_t length,      /* size of string */
```

Lib/test/test_codeccallbacks.py

```
     ...  @@ -1,6 +1,7 @@
1    1    import codecs
2    2    import html.entities
3    3    import itertools
     4  + import re
4    5    import sys
5    6    import unicodedata
6    7    import unittest
     ...  @@ -1124,7 +1125,7 @@ def test_bug828737(self):
1124 1125             text = 'abc<def>ghi'*n
1125 1126             text.translate(charmap)
1126 1127
1127      - def test_mutatingdecodehandler(self):
     1128  + def test_mutating_decode_handler(self):
1128 1129         baddata = [
1129 1130             ("ascii", b"\xff"),
1130 1131             ("utf-7", b"++"),
     ...  @@ -1159,6 +1160,42 @@ def mutating(exc):
1159 1160         for (encoding, data) in baddata:
1160 1161             self.assertEqual(data.decode(encoding, "test.mutating"),
                 "\u4242")
1161 1162
     1163  + def test_mutating_decode_handler_unicode_escape(self):
     1164  +     decode = codecs.unicode_escape_decode
     1165  +     def mutating(exc):
     1166  +         if isinstance(exc, UnicodeDecodeError):
     1167  +             r = data.get(exc.object[:exc.end])
```

```
1168 +                 if r is not None:
1169 +                     exc.object = r[0] + exc.object[exc.end:]
1170 +                     return ('\u0404', r[1])
1171 +             raise AssertionError("don't know how to handle %r" % exc)
1172 +
1173 +         codecs.register_error('test.mutating2', mutating)
1174 +         data = {
1175 +             br'\x0': (b'\\', 0),
1176 +             br'\x3': (b'xxx\\', 3),
1177 +             br'\x5': (b'x\\', 1),
1178 +         }
1179 +         def check(input, expected, msg):
1180 +             with self.assertWarns(DeprecationWarning) as cm:
1181 +                 self.assertEqual(decode(input, 'test.mutating2'), (expected,
      len(input)))
1182 +             self.assertIn(msg, str(cm.warning))
1183 +
1184 +         check(br'\x0n\z', '\u0404\n\\z', r"invalid escape sequence '\z'")
1185 +         check(br'\x0n\501', '\u0404\n\u0141', r"invalid octal escape sequence
      '\501'")
1186 +         check(br'\x0z', '\u0404\\z', r"invalid escape sequence '\z'")
1187 +
1188 +         check(br'\x3n\zr', '\u0404\n\\zr', r"invalid escape sequence '\z'")
1189 +         check(br'\x3zr', '\u0404\\zr', r"invalid escape sequence '\z'")
1190 +         check(br'\x3z5', '\u0404\\z5', r"invalid escape sequence '\z'")
1191 +         check(memoryview(br'\x3z5x')[:-1], '\u0404\\z5', r"invalid escape
      sequence '\z'")
1192 +         check(memoryview(br'\x3z5xy')[:-2], '\u0404\\z5', r"invalid escape
      sequence '\z'")
1193 +
1194 +         check(br'\x5n\z', '\u0404\n\\z', r"invalid escape sequence '\z'")
1195 +         check(br'\x5n\501', '\u0404\n\u0141', r"invalid octal escape sequence
      '\501'")
1196 +         check(br'\x5z', '\u0404\\z', r"invalid escape sequence '\z'")
1197 +         check(memoryview(br'\x5zy')[:-1], '\u0404\\z', r"invalid escape
      sequence '\z'")
1198 +
1162 1199         # issue32583
1163 1200         def test_crashing_decode_handler(self):
1164 1201             # better generating one more character to fill the extra space slot
```

Lib/test/test_codecs.py

```
@@ -1196,23 +1196,39 @@ def test_escape(self):
1196 1196            check(br"[\1010]", b"[A0]")
1197 1197            check(br"[\x41]", b"[A]")
1198 1198            check(br"[\x410]", b"[A0]")
     1199 +
     1200 +    def test_warnings(self):
     1201 +        decode = codecs.escape_decode
     1202 +        check = coding_checker(self, decode)
1199 1203            for i in range(97, 123):
1200 1204                b = bytes([i])
1201 1205                if b not in b'abfnrtvx':
1202      -                 with self.assertWarns(DeprecationWarning):
     1206 +                 with self.assertWarnsRegex(DeprecationWarning,
     1207 +                         r"invalid escape sequence '\\%c'" % i):
1203 1208                      check(b"\\" + b, b"\\" + b)
1204      -             with self.assertWarns(DeprecationWarning):
     1209 +             with self.assertWarnsRegex(DeprecationWarning,
     1210 +                     r"invalid escape sequence '\\%c'" % (i-32)):
1205 1211                  check(b"\\" + b.upper(), b"\\" + b.upper())
1206      -         with self.assertWarns(DeprecationWarning):
     1212 +         with self.assertWarnsRegex(DeprecationWarning,
     1213 +                 r"invalid escape sequence '\\8'"):
1207 1214              check(br"\8", b"\\8")
1208 1215          with self.assertWarns(DeprecationWarning):
1209 1216              check(br"\9", b"\\9")
1210      -         with self.assertWarns(DeprecationWarning):
     1217 +         with self.assertWarnsRegex(DeprecationWarning,
     1218 +                 r"invalid escape sequence '\\\xfa'") as cm:
1211 1219              check(b"\\\xfa", b"\\\xfa")
1212 1220          for i in range(0o400, 0o1000):
1213      -             with self.assertWarns(DeprecationWarning):
     1221 +             with self.assertWarnsRegex(DeprecationWarning,
     1222 +                     r"invalid octal escape sequence '\\%o'" % i):
1214 1223                  check(rb'\%o' % i, bytes([i & 0o377]))
1215 1224
     1225 +         with self.assertWarnsRegex(DeprecationWarning,
     1226 +                 r"invalid escape sequence '\\z'"):
```

```
                1227 +             self.assertEqual(decode(br'\x\z', 'ignore'), (b'\\z', 4))
                1228 +         with self.assertWarnsRegex(DeprecationWarning,
                1229 +                 r"invalid octal escape sequence '\\501'"):
                1230 +             self.assertEqual(decode(br'\x\501', 'ignore'), (b'A', 6))
                1231 +
1216            1232       def test_errors(self):
1217            1233           decode = codecs.escape_decode
1218            1234           self.assertRaises(ValueError, decode, br"\x")
                       @@ -2479,24 +2495,40 @@ def test_escape_decode(self):
2479            2495           check(br"[\x410]", "[A0]")
2480            2496           check(br"\u20ac", "\u20ac")
2481            2497           check(br"\U0001d120", "\U0001d120")
                2498 +
                2499 +     def test_decode_warnings(self):
                2500 +         decode = codecs.unicode_escape_decode
                2501 +         check = coding_checker(self, decode)
2482            2502           for i in range(97, 123):
2483            2503               b = bytes([i])
2484            2504               if b not in b'abfnrtuvx':
2485                    -              with self.assertWarns(DeprecationWarning):
                2505 +                  with self.assertWarnsRegex(DeprecationWarning,
                2506 +                          r"invalid escape sequence '\\%c'" % i):
2486            2507                       check(b"\\" + b, "\\" + chr(i))
2487            2508               if b.upper() not in b'UN':
2488                    -              with self.assertWarns(DeprecationWarning):
                2509 +                  with self.assertWarnsRegex(DeprecationWarning,
                2510 +                          r"invalid escape sequence '\\%c'" % (i-32)):
2489            2511                       check(b"\\" + b.upper(), "\\" + chr(i-32))
2490                    -      with self.assertWarns(DeprecationWarning):
                2512 +         with self.assertWarnsRegex(DeprecationWarning,
                2513 +                 r"invalid escape sequence '\\8'"):
2491            2514               check(br"\8", "\\8")
2492            2515           with self.assertWarns(DeprecationWarning):
2493            2516               check(br"\9", "\\9")
2494                    -      with self.assertWarns(DeprecationWarning):
                2517 +         with self.assertWarnsRegex(DeprecationWarning,
                2518 +                 r"invalid escape sequence '\\\xfa'") as cm:
2495            2519               check(b"\\\xfa", "\\\xfa")
2496            2520           for i in range(0o400, 0o1000):
```

```
2497      -             with self.assertWarns(DeprecationWarning):
          2521  +         with self.assertWarnsRegex(DeprecationWarning,
          2522  +                 r"invalid octal escape sequence '\\%o'" % i):
2498      2523                  check(rb'\%o' % i, chr(i))
2499      2524
          2525  +     with self.assertWarnsRegex(DeprecationWarning,
          2526  +             r"invalid escape sequence '\\z'"):
          2527  +         self.assertEqual(decode(br'\x\z', 'ignore'), ('\\z', 4))
          2528  +     with self.assertWarnsRegex(DeprecationWarning,
          2529  +             r"invalid octal escape sequence '\\501'"):
          2530  +         self.assertEqual(decode(br'\x\501', 'ignore'), ('\u0141', 6))
          2531  +
2500      2532      def test_decode_errors(self):
2501      2533          decode = codecs.unicode_escape_decode
2502      2534          for c, d in (b'x', 2), (b'u', 4), (b'U', 4):
```

...5-05-09-20-22-54.gh-issue-133767.kN2i3Q.rst

```
•••   @@ -0,0 +1,2 @@
1   + Fix use-after-free in the "unicode-escape" decoder with a non-"strict" error
2   + handler.
```

Objects/bytesobject.c

```
      @@ -1048,10 +1048,11 @@ _PyBytes_FormatEx(const char *format, Py_ssize_t
      format_len,
1048  1048    }
1049  1049
1050  1050    /* Unescape a backslash-escaped string. */
1051      -   PyObject *_PyBytes_DecodeEscape(const char *s,
          1051  + PyObject *_PyBytes_DecodeEscape2(const char *s,
1052  1052                                    Py_ssize_t len,
1053  1053                                    const char *errors,
1054      -                                   const char **first_invalid_escape)
          1054  +                                   int *first_invalid_escape_char,
          1055  +                                   const char **first_invalid_escape_ptr)
1055  1056    {
1056  1057        int c;
1057  1058        char *p;
      @@ -1065,7 +1066,8 @@ PyObject *_PyBytes_DecodeEscape(const char *s,
```

```
1065  1066            return NULL;
1066  1067        writer.overallocate = 1;
1067  1068
1068        -     *first_invalid_escape = NULL;
      1069  +     *first_invalid_escape_char = -1;
      1070  +     *first_invalid_escape_ptr = NULL;
1069  1071
1070  1072        end = s + len;
1071  1073        while (s < end) {
            @@ -1103,9 +1105,10 @@ PyObject *_PyBytes_DecodeEscape(const char *s,
1103  1105                        c = (c<<3) + *s++ - '0';
1104  1106                }
1105  1107                if (c > 0377) {
1106        -                   if (*first_invalid_escape == NULL) {
1107        -                       *first_invalid_escape = s-3; /* Back up 3 chars, since
                                we've
1108        -                                                    already incremented s. */
      1108  +                   if (*first_invalid_escape_char == -1) {
      1109  +                       *first_invalid_escape_char = c;
      1110  +                       /* Back up 3 chars, since we've already incremented s. */
      1111  +                       *first_invalid_escape_ptr = s - 3;
1109  1112                    }
1110  1113                }
1111  1114                *p++ = c;
            @@ -1146,9 +1149,10 @@ PyObject *_PyBytes_DecodeEscape(const char *s,
1146  1149                break;
1147  1150
1148  1151            default:
1149        -               if (*first_invalid_escape == NULL) {
1150        -                   *first_invalid_escape = s-1; /* Back up one char, since we've
1151        -                                                already incremented s. */
      1152  +               if (*first_invalid_escape_char == -1) {
      1153  +                   *first_invalid_escape_char = (unsigned char)s[-1];
      1154  +                   /* Back up one char, since we've already incremented s. */
      1155  +                   *first_invalid_escape_ptr = s - 1;
1152  1156                }
1153  1157                *p++ = '\\';
1154  1158                s--;
```

```
@@ -1162,23 +1166,37 @@ PyObject *_PyBytes_DecodeEscape(const char *s,
1162 1166         return NULL;
1163 1167     }
1164 1168
     1169 + // Export for binary compatibility.
     1170 + PyObject *_PyBytes_DecodeEscape(const char *s,
     1171 +                                 Py_ssize_t len,
     1172 +                                 const char *errors,
     1173 +                                 const char **first_invalid_escape)
     1174 + {
     1175 +     int first_invalid_escape_char;
     1176 +     return _PyBytes_DecodeEscape2(
     1177 +             s, len, errors,
     1178 +             &first_invalid_escape_char,
     1179 +             first_invalid_escape);
     1180 + }
     1181 +
1165 1182     PyObject *PyBytes_DecodeEscape(const char *s,
1166 1183                                    Py_ssize_t len,
1167 1184                                    const char *errors,
1168 1185                                    Py_ssize_t Py_UNUSED(unicode),
1169 1186                                    const char *Py_UNUSED(recode_encoding))
1170 1187     {
1171      - -     const char* first_invalid_escape;
1172      - -     PyObject *result = _PyBytes_DecodeEscape(s, len, errors,
1173      - -                                              &first_invalid_escape);
     1188 +     int first_invalid_escape_char;
     1189 +     const char *first_invalid_escape_ptr;
     1190 +     PyObject *result = _PyBytes_DecodeEscape2(s, len, errors,
     1191 +                                               &first_invalid_escape_char,
     1192 +                                               &first_invalid_escape_ptr);
1174 1193         if (result == NULL)
1175 1194             return NULL;
1176      - -     if (first_invalid_escape != NULL) {
1177      - -         unsigned char c = *first_invalid_escape;
1178      - -         if ('4' <= c && c <= '7') {
     1195 +     if (first_invalid_escape_char != -1) {
     1196 +         if (first_invalid_escape_char > 0xff) {
1179 1197                 if (PyErr_WarnFormat(PyExc_DeprecationWarning, 1,
1180      - -                                  "invalid octal escape sequence '\\%.3s'",
```

```
1181      -                                 first_invalid_escape) < 0)
          1198 +                           "invalid octal escape sequence '\\%o'",
          1199 +                           first_invalid_escape_char) < 0)
1182      1200                  {
1183      1201                      Py_DECREF(result);
1184      1202                      return NULL;
          @@ -1187,7 +1205,7 @@ PyObject *PyBytes_DecodeEscape(const char *s,
1187      1205              else {
1188      1206                  if (PyErr_WarnFormat(PyExc_DeprecationWarning, 1,
1189      1207                                       "invalid escape sequence '\\%c'",
1190      -                                    c) < 0)
          1208 +                           first_invalid_escape_char) < 0)
1191      1209                  {
1192      1210                      Py_DECREF(result);
1193      1211                      return NULL;
```

Comments 0