Commit c767536

jrisc committed on Nov 14, 2025

```
Fix DoS vulnerability based on unbounded TCP buffering

In Application.__handle_recv(), the next part of the TCP exchange is
received and queued to the io.BytesIO stream. Then, the content of the
stream was systematically exported to a buffer. However, this buffer
is only used if the data transfer is finished, causing a waste of
processing resources if the message is received in multiple parts.

On top of these unnecessary operations, this function does not handle
length limits properly: it accepts to receive chunks of data with both
an individual and total length larger than the maximum theoretical
length of a Kerberos message, and will continue to wait for data as long
as the input stream's length is not exactly the same as the one provided
in the header of the response (even if the stream is already longer than
the expected length).

If the kdcproxy service is not protected against DNS discovery abuse,
the attacker could take advantage of these problems to operate a
denial-of-service attack (CVE-2025-59089).

After this commit, kdcproxy will interrupt the receiving of a message
after it exceeds the maximum length of a Kerberos message or the length
indicated in the message header. Also it will only export the content of
the input stream to a buffer once the receiving process has ended.

Signed-off-by: Julien Rische <jrische@redhat.com>
```

main (#68) · v1.1.0

1 parent 0606ca5 commit c767536

2 files changed +100 -21 lines changed

Top

Filter files…

kdcproxy

__init__.py

tests.py

2 files changed +100 -21 lines changed

Search within code

kdcproxy/__init__.py

```
@@ -149,6 +149,7 @@ def __await_reply(self, pr, rsocks, wsocks, timeout):
149 149                         if self.sock_type(sock) == socket.SOCK_STREAM:
150 150                             # Remove broken TCP socket from readers
151 151                             rsocks.remove(sock)
    152 +                           read_buffers.pop(sock)
152 153                     else:
153 154                         if reply is not None:
154 155                             return reply
@@ -174,7 +175,7 @@ def __handle_recv(self, sock, read_buffers):
174 175             if self.sock_type(sock) == socket.SOCK_DGRAM:
175 176                 # For UDP sockets, recv() returns an entire datagram
176 177                 # package. KDC sends one datagram as reply.
177     -               reply = sock.recv(1048576)
    178 +               reply = sock.recv(self.MAX_LENGTH)
178 179                 # If we proxy over UDP, we will be missing the 4-byte
179 180                 # length prefix. So add it.
180 181                 reply = struct.pack("!I", len(reply)) + reply
@@ -186,30 +187,38 @@ def __handle_recv(self, sock, read_buffers):
186 187             if buf is None:
187 188                 read_buffers[sock] = buf = io.BytesIO()
188 189
189     -           part = sock.recv(1048576)
190     -           if not part:
191     -               # EOF received.  Return any incomplete data we have on the theory
192     -               # that a decode error is more apparent than silent failure.  The
193     -               # client will fail faster, at least.
194     -               read_buffers.pop(sock)
195     -               reply = buf.getvalue()
196     -               return reply
    190 +           part = sock.recv(self.MAX_LENGTH)
    191 +           if part:
    192 +               # Data received, accumulate it in a buffer.
    193 +               buf.write(part)
197 194
```

```
198       -         # Data received, accumulate it in a buffer.
199       -         buf.write(part)
      195 +             reply = buf.getbuffer()
      196 +             if len(reply) < 4:
      197 +                 # We don't have the length yet.
      198 +                 return None
200   199
201       -         reply = buf.getvalue()
202       -         if len(reply) < 4:
203       -             # We don't have the length yet.
204       -             return None
      200 +             # Got enough data to check if we have the full package.
      201 +             (length, ) = struct.unpack("!I", reply[0:4])
      202 +             length += 4  # add prefix length
205   203
206       -         # Got enough data to check if we have the full package.
207       -         (length, ) = struct.unpack("!I", reply[0:4])
208       -         if length + 4 == len(reply):
209       -             read_buffers.pop(sock)
210       -             return reply
      204 +             if length > self.MAX_LENGTH:
      205 +                 raise ValueError('Message length exceeds the maximum length '
      206 +                                  'for a Kerberos message (%i > %i)'
      207 +                                  % (length, self.MAX_LENGTH))
211   208
212       -         return None
      209 +             if len(reply) > length:
      210 +                 raise ValueError('Message length exceeds its expected length '
      211 +                                  '(%i > %i)' % (len(reply), length))
      212 +
      213 +             if len(reply) < length:
      214 +                 return None
      215 +
      216 +         # Else (if part is None), EOF was received.  Return any incomplete data
      217 +         # we have on the theory that a decode error is more apparent than
      218 +         # silent failure.  The client will fail faster, at least.
      219 +
      220 +         read_buffers.pop(sock)
      221 +         return buf.getvalue()
213   222
```

```
214  223        def __filter_addr(self, addr):
215  224            if addr[0] not in (socket.AF_INET, socket.AF_INET6):
```

tests.py

```
@@ -20,6 +20,8 @@
20  20   # THE SOFTWARE.
21  21
22  22   import os
    23 + import socket
    24 + import struct
23  25   import unittest
24  26   from base64 import b64decode
25  27   try:
@@ -122,6 +124,74 @@ def test_no_server(self):
122 124                                                   kpasswd=True)
123 125            self.assertEqual(response.status_code, 503)
124 126
    127 +    @mock.patch("socket.getaddrinfo", return_value=addrinfo)
    128 +    @mock.patch("socket.socket")
    129 +    def test_tcp_message_length_exceeds_max(self, m_socket, m_getaddrinfo):
    130 +        # Test that TCP messages with length > MAX_LENGTH raise ValueError
    131 +        # Create a message claiming to be larger than MAX_LENGTH
    132 +        max_len = self.app.MAX_LENGTH
    133 +        # Length prefix claiming message is larger than allowed
    134 +        oversized_length = max_len + 1
    135 +        malicious_msg = struct.pack("!I", oversized_length)
    136 +
    137 +        # Mock socket to return the malicious length prefix
    138 +        mock_sock = m_socket.return_value
    139 +        mock_sock.recv.return_value = malicious_msg
    140 +        mock_sock.getsockopt.return_value = socket.SOCK_STREAM
    141 +
    142 +        # Manually call the receive method to test it
    143 +        read_buffers = {}
    144 +        with self.assertRaises(ValueError) as cm:
    145 +            self.app._Application__handle_recv(mock_sock, read_buffers)
    146 +
    147 +        self.assertIn("exceeds the maximum length", str(cm.exception))
```

```
148 +         self.assertIn(str(max_len), str(cm.exception))
149 +
150 +     @mock.patch("socket.getaddrinfo", return_value=addrinfo)
151 +     @mock.patch("socket.socket")
152 +     def test_tcp_message_data_exceeds_expected_length(
153 +         self, m_socket, m_getaddrinfo
154 +     ):
155 +         # Test that receiving more data than expected raises ValueError
156 +         # Create a message with length = 100 but send more data
157 +         expected_length = 100
158 +         length_prefix = struct.pack("!I", expected_length)
159 +         # Send more data than the length prefix indicates
160 +         extra_data = b"X" * (expected_length + 10)
161 +         malicious_msg = length_prefix + extra_data
162 +
163 +         mock_sock = m_socket.return_value
164 +         mock_sock.recv.return_value = malicious_msg
165 +         mock_sock.getsockopt.return_value = socket.SOCK_STREAM
166 +
167 +         read_buffers = {}
168 +         with self.assertRaises(ValueError) as cm:
169 +             self.app._Application__handle_recv(mock_sock, read_buffers)
170 +
171 +         self.assertIn("exceeds its expected length", str(cm.exception))
172 +
173 +     @mock.patch("socket.getaddrinfo", return_value=addrinfo)
174 +     @mock.patch("socket.socket")
175 +     def test_tcp_eof_returns_buffered_data(self, m_socket, m_getaddrinfo):
176 +         # Test that EOF returns any buffered data
177 +         initial_data = b"\x00\x00\x00\x10"  # Length = 16
178 +         mock_sock = m_socket.return_value
179 +         mock_sock.getsockopt.return_value = socket.SOCK_STREAM
180 +
181 +         # First recv returns some data, second returns empty (EOF)
182 +         mock_sock.recv.side_effect = [initial_data, b""]
183 +
184 +         read_buffers = {}
185 +         # First call buffers the data
186 +         result = self.app._Application__handle_recv(mock_sock, read_buffers)
187 +         self.assertIsNone(result)  # Not complete yet
```

```
        188 +
        189 +         # Second call gets EOF and returns buffered data
        190 +         result = self.app._Application__handle_recv(mock_sock, read_buffers)
        191 +         self.assertEqual(result, initial_data)
        192 +         # Buffer should be cleaned up
        193 +         self.assertNotIn(mock_sock, read_buffers)
        194 +
125     195
126     196     def decode(data):
127     197         data = data.replace(b'\\n', b'')
```

Comments 0

Please sign in to comment.