TDTS06 Laboration 2

Gunnar Grimsdal (gungr890)
Daniel Månsson (danma344)

September 2017

1 Manual

The proxy is written in Python3 (meaning that it will require Python3.x to be installed on the computer), this means that running the proxy is not harder than executing "server.py" with Python3 together with an optional port parameter (port 13337 will be used by default if no parameter is supplied). This example would start the proxy on port 8080:

```
$ python3 server.py 8080
```

The three necessary Python-files can be downloaded from:
`https://github.com/guggenn/TDTS06-Proxy`

2 Features

The proxy supports both HTTP/1.0 and HTTP/1.1 with the only exception that even for HTTP/1.0 the "Host:" field MUST be set in the request header. Detection of inappropriate content within a Web page only works for plain text, meaning that it does not work for data that is compressed with GZIP or that is chunked (however it does partially work for chunked data as long as the filtered word is not separated in multiple chunks).

The URL-filtering of the proxy can be seen in the "connection_handler" function located in "main.py", this function calls another function named "is_acceptable" that is located in the file "request.py". This function extracts the URL from the first line of the request and checks the URL for unacceptable keywords. If any unacceptable keywords are located in the URL the function "connection_handler" will create a new responses with HTTP status code 301 and the header field **Location:** with the value of the error page.

The content-filtering of the proxy can be seen in the function "send_and_receive" located in the "main.py" file, this code calls a function named "is_acceptable" after checking if the content of the response is plain text. The function "is_acceptable" which is located in "response.py" loops over all unacceptable keywords

and checks if they exist in the payload. If any unacceptable keywords are detected in the payload the "send_and_receive" function will create a new responses with HTTP status code 301 and the header field **Location:** with the value of the error page.

The proxy should be able to handle all the following different header types (GET, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE & PATCH) but there has not been any real testing outside of the GET, POST and OPTIONS.

One key features of the proxy is that it runs on multiple threads. When a new client connects to the server a new thread is created. This prevents one faulty connection to a server from slowing down or hanging other connections. This is very important due to the absence of timeouts on the connections.

Connections that demands SSL-encryption can't be used with this proxy. Websites using GZIP should work properly also downloading images and other non-text payloads should work, although without any type of content filtering.

One unusual feature of the proxy is that we on every request from the client change the header **Connection:** field to the "Close". This is done to avoid the handling of persistent connections which are used multiple times before closing, which in turn would make put more requirements on how the proxy handles connections.

Source Code

Server.py

```
import socket
import sys
from request import Request
from response import Response
import _thread as thread

BUFFER_SIZE = 1024
REQ_RED_URL = b'http://www.ida.liu.se/~TDTS04/labs/2011/ass2/error1.html'
RESP_RED_URL = b'http://www.ida.liu.se/~TDTS04/labs/2011/ass2/error2.html'


# Returns a tuple with the header as first element and eventually a part
# of the body (if it reads too much) as the second element
def read_header(sock):
    data = b''
    fragment = b''
    end_sequence = -1
    while True:
        fragment = sock.recv(BUFFER_SIZE)
        if not fragment:
            break
        data += fragment
        end_sequence = data.find(b'\r\n\r\n')
        if end_sequence != -1:
            end_sequence += 4
            break
    return data[:end_sequence], data[end_sequence:]


# Read the HTTP payload from a request or an response
def read_body(sock, length):
    data_size = 0
    data = b''
    fragment = b''
    while data_size < length:
        fragment = sock.recv(length - data_size)
        if fragment == b'':
            break
        data += fragment
        data_size += len(fragment)
    return data

```

```
43
44  # Read the HTTP payload in request or response that uses chunked data
45  def read_chunked_body(sock, msg_part=b''):
46      END = b'\r\n'
47      current_index = 0
48      next_data_size = 0
49      next_chunk_size_byte = b''
50      data = msg_part
51
52      while True:
53          if len(data) == 0:
54              data += sock.recv(BUFFER_SIZE)
55          if END in data[current_index:]:
56              end_index = current_index+data[current_index:].find(END)
57              next_chunk_size_byte = data[current_index:end_index]
58              next_data_size = int(next_chunk_size_byte, 16)
59              if next_data_size == 0:
60                  break
61              current_index += next_data_size + len(next_chunk_size_byte) + 4
62          data += sock.recv(BUFFER_SIZE)
63      return data
64
65
66  # Send data to a server and return a response
67  def send_and_receive(msg):
68      server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
69      # Send client message to server..
70      print("Sending to {}".format(msg.get_host()))
71
72      # Connecting to webserver
73      address, port = msg.get_host()
74      #If no port or address exists there will be a None value returned
75      port = port if port else 80
76      address = address if address else msg.get_URL()
77      server_socket.connect((address, port)) # TODO Don't assume port 80?
78
79      # Forward the request to the server
80      server_socket.sendall(msg.byte_data)
81
82      # Read response from server
83      header, payload = read_header(server_socket)
84      server_msg = Response(header)
85      con_leng = server_msg.get_header_element(b'Content-Length:')
86      # If the package contains a payload read it
87      if con_leng:
88          con_leng = int(con_leng) - len(payload)
```

```
89          server_msg.add(payload + read_body(server_socket, int(con_leng)))
90      # If the response is chunked
91      elif server_msg.get_header_element(b'Transfer-Encoding:') == b'chunked':
92          server_msg.add(read_chunked_body(server_socket, payload))
93
94      server_socket.close() # Done with socket
95      # This is where the content is checked for unaccable keywords
96      # If not acceptable request redirect to error page
97      if server_msg.is_text() and not server_msg.is_acceptable():
98          server_msg = Response(b'HTTP/1.1 301 Moved Permanently\r\n\r\n')
99          server_msg.set_header_element(b'Location:', RESP_RED_URL)
100
101     return server_msg
102
103
104 # Read the request from the client
105 def read_client(sock):
106     header, payload = read_header(sock)
107     client = Request(header)
108     # Get Content-Length to know how how many bites to read from socket
109     con_leng = client.get_header_element(b'Content-Length:')
110     # If there is a Content-Length read so many bytes
111     if con_leng:
112         con_leng = int(con_leng) - len(payload)
113         client.add(payload + read_body(sock, int(con_leng)))
114     # If the request is chunked
115     elif client.get_header_element(b'Transfer-Encoding:') == b'chunked':
116         client.add(read_chunked_body(sock, payload))
117     return client
118
119
120 # Run for every new connection
121 def connection_handler(client):
122     client_msg = read_client(client)
123     if len(client_msg.byte_data) < 4: # this is a bad request
124         return
125     # Check if the words that should be filterd are not in the url
126     if client_msg.is_acceptable():
127         client_msg.set_header_element(b'Connection:', b'close')
128         server_msg = send_and_receive(client_msg)
129     # This is where the URL is checked for unaccable keywords
130     else:  # If not acceptable request redirect to error page
131         server_msg = Response(b'HTTP/1.1 301 Moved Permanently\r\n\r\n')
132         server_msg.set_header_element(b'Location:', REQ_RED_URL)
133     client.sendall(server_msg.byte_data)
134     client.close()
```

```
135
136
137 # The main functions starts a socket
138 def main():
139     # If port number is not in de exec use port 13337
140     port = 13337 if len(sys.argv) < 2 else int(sys.argv[1])
141
142     ip_addr = '127.0.0.1' # Ip addess to listen to
143     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
144     # Close socket on thread close
145     sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
146     sock.bind((ip_addr, port))
147     sock.listen(1)
148     print("Server started at {}:{}".format(ip_addr, str(port)))
149     while True:
150         client, address = sock.accept() # Client connected
151         # Start a thread for the new client
152         thread.start_new_thread(connection_handler, (client, ))
153
154
155 if __name__ == '__main__':
156     main()
```

Request.py

```
FILTER_URL_TAGS = [
    b'SpongeBob', b'Britney Spears', b'Paris Hilton', b'Norrk??ping'
]


class Request:
    # Return a list of tuples
    # Example [(header_element_name, header_element_value), ...]
    def get_header_data(self):
        header_data = []
        for data in self.get_header().split(b'\r\n'):
            header_data.append(
                (data.split(b" ")[0], b' '.join(data.split(b' ')[1:])))
        return header_data

    # Get header as byte string
    def get_header(self):
        return self.byte_data.split(b'\r\n\r\n')[0]

    # Get payload as byte string. Used if post method
    def get_payload(self):
        if len(self.byte_data.split(b'\r\n\r\n')) > 1:
            return self.byte_data.split(b'\r\n\r\n')[1]
        else:
            return b''

    # Get value from a specific header element
    def get_header_element(self, header_name):
        for element_name, element in self.get_header_data():
            if element_name == header_name:
                return element
        return None

    # Add data to the internal byte string
    def add(self, msg):
        self.byte_data += msg

    # Get HOST as a tuple of (address, port),
    # where any can be None due to HTTP version or lack of port number
    def get_host(self):
        value = self.get_header_element(b'Host:')
        #Check if it's empty, if so return None, None
        if not value:
            return None, None
```

```
45
46          # Return the UTF-8 string if it exists else None
47          if b':' in value:
48              value = value.split(b':')
49              return value[0].decode('utf-8'), int(value[1])
50          return value.decode('utf-8'), None
51
52      # Return the first element of the header and it's value
53      # This should always be the url
54      def get_URL(self):
55          return self.get_header_data()[0][1].split(b' ')[0]
56      # Return True if the request is a GET request
57      def is_get(self):
58          return self.get_header_element(b'GET') is None
59
60      # Return True if the request is a GET request
61      def is_post(self):
62          return self.get_header_element(b'POST') is None
63
64      # Set the payload in the internal byte string
65      def set_payload(self, payload):
66          self.byte_data = self.get_header() + b'\r\n\r\n' + payload
67
68      # Set the header element name and value in form of byte strings
69      # and add them to the byte string self.byte_data
70      def set_header_element(self, header_name, header_value):
71          header = self.get_header_data()
72          # Loop over all the element and see if the header_name already exists
73          for index in range(len(header)):
74              if header[index][0] == header_name:
75                  # The header exists change the value and save to byte_data
76                  header[index] = (header_name, header_value)
77                  self.__set_header(header)
78                  return
79          # If the element is not yet in the header add it now
80          header.append((header_name, header_value))
81          self.__set_header(header)
82
83      # Get a list with the data and write it to the
84      # byte string self.byte_data
85      def __set_header(self, data_dict):
86          local_header = b''
87          for key, value in data_dict:
88              local_header += key + b' ' + value + b'\r\n'
89          self.byte_data = local_header + b'\r\n' + self.get_payload()
90
```

```
91      #  Check if not accepted are in the payload
92      def is_acceptable(self):
93          for tag in FILTER_URL_TAGS:
94              # If a tag is found the content is not safe
95              if tag in self.get_URL():
96                  return False
97          return True # No tags were found
98
99      def __init__(self, byte_data):
100         self.byte_data = byte_data
```

Response.py

```
FILTER_TAGS = [
    b'SpongeBob', b'Britney Spears', b'Paris Hilton', b'Norrk&oumlping'
]


class Response:

    # Return a list of tuples
    # Example [(header_element_name, header_element_value), ...]
    def get_header_list(self):
        header_data = []
        for data in self.get_header().split(b'\r\n'):
            header_data.append(
                (data.split(b" ")[0], b' '.join(data.split(b' ')[1:])))
        return header_data

    # Get header as byte string
    def get_header(self):
        return self.byte_data.split(b'\r\n\r\n')[0]

    # Get payload as a byte string (HTML or file)
    def get_payload(self):
        if len(self.byte_data.split(b'\r\n\r\n')) > 1:
            return self.byte_data.split(b'\r\n\r\n')[1]
        else:
            return None

    # Get value from a specific header element
    def get_header_element(self, header_name):
        for element_name, element in self.get_header_list():
            if element_name == header_name:
                return element
        return None  # If Host: not found

    # Return true if the Content-Type starts with text
    def is_text(self):
        con_typ = self.get_header_element(b'Content-Type:')
        return con_typ .startswith(b'text') if con_typ else False

    # Add data to the internal byte string
    def add(self, msg):
        self.byte_data += msg

    # Set the header element name and value in form of byte strings
```

```
45      # and add theme to the byte string self.byte_data
46      def set_header_element(self, header_name, header_value):
47          header = self.get_header_list()
48          for index in range(len(header)):
49              if header[index][0] == header_name:
50                  header[index] = (header_name, header_value)
51                  self.__set_header(header)
52                  return
53          # If the element is not yet in the header add it now
54          header.append((header_name, header_value))
55          self.__set_header(header)
56
57      # Get a list with the data and write it to the
58      # byte string self.byte_data
59      def __set_header(self, data_dict):
60          local_header = b''
61          for key, value in data_dict:
62              local_header += key + b' ' + value + b'\r\n'
63          self.byte_data = local_header + b'\r\n' + self.get_payload()
64
65      # Sets the payload
66      def set_payload(self, payload):
67          self.byte_data = self.get_header() + b'\r\n\r\n' + payload
68
69      # Checks if the payload contains any forbidden keywords
70      def is_acceptable(self):
71          for tag in FILTER_TAGS:
72              if tag in self.get_payload():
73                  return False
74          return True
75
76      def __init__(self, byte_data=b''):
77          self.byte_data = byte_data
```