OpenVPN

- Diagnose / Fehlersuche
  - OpenVPN Check Server online Script

Diagnose / Fehlersuche

OpenVPN Check Server online Script

Beschreibung:

Ein Python Script das überprüft ob der openvpn server läuft.

Aufrufen:

```
#als udp test
python3 check_vpn.py -p 1194 ipadresse/hostname
#als tcp test, dafür -t
python3 check_vpn.py -p 443 -t ipadresse/hostname
```

Quelltext:

Auch zum download hier

```
#!/usr/bin/env python
#
# Check if an OpenVPN server runs on a given UDP or TCP port.
#

import os
import sys
import time
import hmac
import hashlib
import struct
import socket
import argparse
import binascii
```

```
P_CONTROL_HARD_RESET_CLIENT_V2 = 7
P_CONTROL_HARD_RESET_SERVER_V2 = 8
MAX_P_CONTROL_SIZE = 128 # TODO: adopt
MAX_CIPHER_KEY_LENGTH = 64
MAX_HMAC_KEY_LENGTH = 64
BUFFER_SIZE = 1024

ALGORITHMS_AVAILABLE = hashlib.algorithms_available if hasattr(hashlib, "algorithms_available") else
hashlib.algorithms

def ok(msg):
    print('OK: %s' % msg)
    return 0

def warning(msg):
    print('WARN: %s' % msg)
    return 1

def critical(msg):
    print('CRIT: %s' % msg)
    return 1

def build_p_control_hard_reset_client_v2(sid, digest, key):
    # see openvpn source code "src/openvpn/crypto.c" function openvpn_decrypt_v1
    pid = 1 # packet id
    ts = int(time.time())  # net time

    if key:
        # generate hmac
        h = hmac.new(key, digestmod=digest)
        h.update(struct.pack('>I', pid)) # packet id
        h.update(struct.pack('>I', ts)) # net time
        h.update(struct.pack('>B', P_CONTROL_HARD_RESET_CLIENT_V2 << 3)) # packet type
        h.update(sid) # session id
        h.update(struct.pack('>B', 0)) # message packet id array length
        h.update(struct.pack('>I', 0)) # message packet id

    # build packet
    result = b''
```

```
    result += struct.pack('>B', P_CONTROL_HARD_RESET_CLIENT_V2 << 3) # packet type
    result += sid # session id
    if key: result += h.digest() # hmac
    if key: result += struct.pack('>I', pid) # packet id
    if key: result += struct.pack('>I', ts) # net time
    result += struct.pack('>B', 0) # message packet id array length
    result += struct.pack('>I', 0) # message packet id
    return result

def validate_p_control_hard_reset_server_v2(packet, query_sid, digest, digest_size, key):
    # see openvpn source code "src/openvpn/crypto.c" function openvpn_decrypt_v1
    # identify packet
    if False: pass
    elif len(packet) - struct.unpack('>B', packet[9:10])[0] * 4 == 14: plen = 0 # type sid mpida mpid
    elif len(packet) - struct.unpack('>B', packet[9:10])[0] * 4 == 22: plen = 1 # type sid mpida rsid mpid
    elif len(packet) - struct.unpack('>B', packet[17+digest_size:18+digest_size])[0] * 4 == 30+digest_size: plen
= 2 # type sid hmac pid ts mpida rsid mpid
    else: return 20

    # parse packet
    ptype = struct.unpack('>B', packet[:1])[0] # packet type
    packet = packet[1:]
    sid = packet[:8] # session id
    packet = packet[8:]
    if plen >= 2:
        phmac = packet[:digest_size] # hmac
        packet = packet[digest_size:]
        pid = struct.unpack('>I', packet[:4])[0] # packet id
        packet = packet[4:]
        ts = struct.unpack('>I', packet[:4])[0] # net time
        packet = packet[4:]
    mpidlen = struct.unpack('>B', packet[:1])[0] # message packet id array length
    packet = packet[1:]
    mpidarray = [] # packet id array
    for i in range(mpidlen):
        mpidarray.append(struct.unpack('>I', packet[:4])[0]) # message packet id array element
        packet = packet[4:]
    if plen >= 1:
        rsid = packet[:8] # remote session id
        packet = packet[8:]
```

```
    mpid = struct.unpack('>I', packet[:4])[0] # message packet id

    # validate packet
    if ptype != P_CONTROL_HARD_RESET_SERVER_V2 << 3: return 20
    if mpid != 0: return 20
    if plen >= 1 and rsid != query_sid: return 20
    if plen >= 2 and key:
        if pid != 1: return 20
        # generate hmac
        h = hmac.new(key, digestmod=digest)
        h.update(struct.pack('>I', pid)) # packet id
        h.update(struct.pack('>I', ts)) # net time
        h.update(struct.pack('>B', ptype)) # packet type
        h.update(sid) # session id
        h.update(struct.pack('>B', mpidlen)) # message packet id array length
        for e in mpidarray: h.update(struct.pack('>I', e)) # message packet id array element
        h.update(rsid) # remote session id
        h.update(struct.pack('>I', mpid)) # message packet id
        if phmac != h.digest(): return 10
        return 1
    return 0

def check(host, port, tcp, timeout, digest, digest_size, client_key, server_key, retrycount, validate):
    sid = os.urandom(8) # session id
    packet = build_p_control_hard_reset_client_v2(sid, digest, client_key)
    query_server = query_tcp_server if tcp else query_udp_server

    try:
        s = create_socket(host, port, tcp, timeout)
    except socket.error:
        return critical('Unable to create socket')

    try:
        response = query_server(s, host, port, packet, retrycount)
    except RuntimeError:
        return critical('Invalid response')
    except:
        return critical('Not responding')
    finally:
        s.close()
```

```
    if response is None:
        return critical('Not responding')

    # for debugging purpose
    # response = binascii.hexlify(response)
    if not validate: return ok('Responded')
    valid = validate_p_control_hard_reset_server_v2(response, sid, digest, digest_size, server_key)
    if valid == 0: return ok('Response validated')
    if valid == 1: return ok('Response validated, checked HMAC')
    if valid == 10: return warning('Invalid HMAC')
    return critical('Invalid response')

def create_socket(host, port, tcp, timeout):
    # thanks to glucas for the idea
    sock_type = socket.SOCK_STREAM if tcp else socket.SOCK_DGRAM
    af, socktype, proto, canonname, sa = socket.getaddrinfo(host, port, socket.AF_UNSPEC, sock_type)[0]
    s = socket.socket(af, socktype, proto)
    s.settimeout(timeout)
    return s

def query_udp_server(s, host, port, packet, retrycount):
    # Send up to 'retrycount' UDP packets with 'timeout' secs between each.
    # Return data after receiving first UDP packet.
    for i in range(retrycount):
        try:
            s.sendto(packet, (host, port))
            # TODO: check response address
            data, _ = s.recvfrom(BUFFER_SIZE)
            return data
        except socket.timeout:
            pass
        except:
            return None
    return None

def query_tcp_server(s, host, port, packet, retrycount):
    # ignore retrycount
    s.connect((host, port))
    s.send(struct.pack('>H', len(packet)) + packet)
```

```
    length = struct.unpack('>H', s.recv(2))[0]
    if length > MAX_P_CONTROL_SIZE: raise RuntimeError
    data = s.recv(length)
    if len(data) != length: raise RuntimeError
    return data

def readkey(path):
    key = None
    try:
        with open(path, 'r') as myfile: key = myfile.read()
    except:
        return None
    index_start = key.find('-\n');
    index_end = key.find('\n-', index_start);
    if index_start < 0 or index_end < 0 or index_end <= index_start:
        return None
    index_start += 2
    key = key[index_start:index_end].replace('\n', '').replace('\r', '')
    key = binascii.unhexlify(key)
    return key

def optionsparser(argv=None):
    parser = argparse.ArgumentParser()
    parser.add_argument('-p', '--port', help='set port number (default is %(default)d)', type=int, default=1194)
    parser.add_argument('-t', '--tcp', help='use tcp instead of udp', action='store_true')
    parser.add_argument('--timeout', help='set timeout in seconds, for udp counted per packet (default is
%(default)d)', type=int, default=2)
    parser.add_argument('--digest', help='set digest algorithm (default is "%(default)s")', default='sha1')
    parser.add_argument('--digest-size', help='set HMAC digest size', type=int)
    parser.add_argument('--digest-key-client', help='set client HMAC key')
    parser.add_argument('--digest-key-server', help='set server HMAC key for packet validation')
    parser.add_argument('--tls-auth', help='set tls-auth file')
    # TODO: direction argument (normal, inverse)
    parser.add_argument('--tls-auth-inverse', help='set tls-auth file direction to inverse (1)', action='store_true')
    parser.add_argument('--retrycount', help='number of udp retries before giving up (default is %(default)d)',
type=int, default=3)
    parser.add_argument('--no-validation', help='do not validate response', action='store_true')
    parser.add_argument('host', help='the OpenVPN host name or IP')
    return parser.parse_args(argv)
```

```
def main(argv=None):
    args = optionsparser(argv)

    if args.digest_size and args.digest_size < 0:
        critical('digest size must be positive')
    if args.retrycount < 1:
        critical('retry count must be positive')
    if args.tls_auth and (args.digest_key_client or args.digest_key_server):
        critical('--tls-auth cannot go with --digest-key')

    client_key = binascii.unhexlify(args.digest_key_client) if args.digest_key_client else None
    server_key = binascii.unhexlify(args.digest_key_server) if args.digest_key_server else None
    digest = args.digest
    digest_size = args.digest_size

    digest = digest.lower()
    if digest not in ALGORITHMS_AVAILABLE:
        return critical('digest not available')
    try:
        digest = getattr(hashlib, digest)
        if not digest_size: digest_size = digest().digest_size
    except:
        return critical('digest creation failed')

    if args.tls_auth:
        # see openvpn source code "src/openvpn/crypto.h", "src/openvpn/crypto.c" and
"src/openvpn/crypto_backend.h"
        # 64 byte cipher direction 0, 64 byte hmac direction 0, 64 byte cipher direction 1, 64 byte hmac direction 1
(=> 2048 bit)
        key = readkey(args.tls_auth)
        if key == None: return critical('cannot read tls auth file')
        index_start = MAX_CIPHER_KEY_LENGTH + MAX_HMAC_KEY_LENGTH + MAX_CIPHER_KEY_LENGTH
        index_end = index_start + MAX_HMAC_KEY_LENGTH
        client_key = key[index_start:index_end]
        index_start = MAX_CIPHER_KEY_LENGTH
        index_end = index_start + MAX_HMAC_KEY_LENGTH
        server_key = key[index_start:index_end]
        if args.tls_auth_inverse: client_key, server_key = server_key, client_key
        # reduce key size to required size
        client_key = client_key[:digest_size]
```

```
        server_key = server_key[:digest_size]

    return check(args.host, args.port, args.tcp, args.timeout, digest, digest_size, client_key, server_key,
args.retrycount, not args.no_validation)

if __name__ == '__main__':
    code = main()
    sys.exit(code)
```