"""

Simple HTTP Live Streaming client.

References:

http://tools.ietf.org/html/draft-pantos-http-live-streaming-08

This program is free software. It comes without any warranty, to

the extent permitted by applicable law. You can redistribute it

and/or modify it under the terms of the Do What The Fuck You Want

To Public License, Version 2, as published by Sam Hocevar. See

http://sam.zoy.org/wtfpl/COPYING for more details.

Last updated: July 22, 2012

MODIFIED BY shani to make it work with F4mProxy

"""

import urlparse, urllib2, subprocess, os,traceback,cookielib,re,Queue,threading

import xml.etree.ElementTree as etree

import base64

from struct import unpack, pack

import struct

import sys

import io

import os

import time

import itertools

import xbmcaddon

import xbmc

import urllib2,urllib

import traceback

import urlparse

import posixpath

import re

import hmac

import hashlib

import binascii

import zlib

from hashlib import sha256

import cookielib

import array, random, string

import requests

#from Crypto.Cipher import AES

'''

from crypto.cipher.aes import AES

from crypto.cipher.cbc import CBC

from crypto.cipher.base import padWithPadLen

from crypto.cipher.rijndael import Rijndael

from crypto.cipher.aes\_cbc import AES\_CBC

'''

gproxy=None

gauth=None

nsplayer=False

callbackDRM=None

try:

from Crypto.Cipher import AES

USEDec=1 ## 1==crypto 2==local, local pycrypto

except:

print 'pycrypt not available using slow decryption'

USEDec=3 ## 1==crypto 2==local, local pycrypto

if USEDec==1:

#from Crypto.Cipher import AES

print 'using pycrypto'

elif USEDec==2:

from decrypter import AESDecrypter

AES=AESDecrypter()

else:

from f4mUtils import python\_aes

#from decrypter import AESDecrypter

iv=None

key=None

value\_unsafe = '%+&;#'

VALUE\_SAFE = ''.join(chr(c) for c in range(33, 127)

if chr(c) not in value\_unsafe)

SUPPORTED\_VERSION=3

cookieJar=cookielib.LWPCookieJar()

clientHeader=None

class HLSDownloaderRetry():

global cookieJar

"""

A downloader for f4m manifests or AdobeHDS.

"""

def \_\_init\_\_(self):

self.init\_done=False

def init(self, out\_stream, url, proxy=None,use\_proxy\_for\_chunks=True,g\_stopEvent=None, maxbitrate=0, auth='', callbackpath="", callbackparam=""):

global clientHeader,gproxy,gauth

try:

self.init\_done=False

self.init\_url=url

clientHeader=None

self.status='init'

self.proxy = proxy

self.auth=auth

self.callbackpath=callbackpath

self.callbackparam=callbackparam

if self.auth ==None or self.auth =='None' or self.auth=='':

self.auth=None

if self.auth:

gauth=self.auth

if self.proxy and len(self.proxy)==0:

self.proxy=None

gproxy=self.proxy

self.use\_proxy\_for\_chunks=use\_proxy\_for\_chunks

self.out\_stream=out\_stream

if g\_stopEvent: g\_stopEvent.clear()

self.g\_stopEvent=g\_stopEvent

self.maxbitrate=maxbitrate

if '|' in url:

sp = url.split('|')

url = sp[0]

clientHeader = sp[1]

print clientHeader

clientHeader= urlparse.parse\_qsl(clientHeader)

print 'header recieved now url and headers are',url, clientHeader

self.status='init done'

self.url=url

return True# disabled downloadInternal(self.url,None,self.maxbitrate,self.g\_stopEvent , self.callbackpath, self.callbackparam, testing=True)

except:

traceback.print\_exc()

self.status='finished'

return False

def keep\_sending\_video(self,dest\_stream, segmentToStart=None, totalSegmentToSend=0):

try:

self.status='download Starting'

downloadInternal(self.url,dest\_stream,self.maxbitrate,self.g\_stopEvent , self.callbackpath, self.callbackparam)

except:

traceback.print\_exc()

print 'setting finished'

self.status='finished'

def getUrl(url,timeout=15,returnres=False,stream =False):

global cookieJar

global clientHeader

global nsplayer

try:

post=None

print 'url',url

session = requests.Session()

session.cookies = cookieJar

headers = {'User-Agent': 'Mozilla/5.0 (X11; Linux i686; rv:42.0) Gecko/20100101 Firefox/42.0 Iceweasel/42.0'}

if clientHeader:

for n,v in clientHeader:

headers[n]=v

if nsplayer:

print 'nsplayer is true'

headers['User-Agent']=binascii.b2a\_hex(os.urandom(20))[:32]

print 'nsplayer', nsplayer,headers

proxies={}

if gproxy:

proxies= {"http": "http://"+gproxy}

#import random

#headers['User-Agent'] =headers['User-Agent'] + str(int(random.random()\*100000))

if post:

req = session.post(url, headers = headers, data= post, proxies=proxies,verify=False,timeout=timeout,stream=stream)

else:

req = session.get(url, headers=headers,proxies=proxies,verify=False ,timeout=timeout,stream=stream)

req.raise\_for\_status()

if returnres:

return req

else:

return req.text

except:

print 'Error in getUrl'

traceback.print\_exc()

raise

return None

def getUrlold(url,timeout=20, returnres=False):

global cookieJar

global clientHeader

try:

post=None

#print 'url',url

#openner = urllib2.build\_opener(urllib2.HTTPHandler, urllib2.HTTPSHandler)

cookie\_handler = urllib2.HTTPCookieProcessor(cookieJar)

openner = urllib2.build\_opener(cookie\_handler, urllib2.HTTPBasicAuthHandler(), urllib2.HTTPHandler())

#print cookieJar

if post:

req = urllib2.Request(url, post)

else:

req = urllib2.Request(url)

ua\_header=False

if clientHeader:

for n,v in clientHeader:

req.add\_header(n,v)

if n=='User-Agent':

ua\_header=True

if not ua\_header:

req.add\_header('User-Agent','AppleCoreMedia/1.0.0.12B411 (iPhone; U; CPU OS 8\_1 like Mac OS X; en\_gb)')

#req.add\_header('X-Playback-Session-Id','9A1E596D-6AB6-435F-85D1-59BDD0E62D24')

if gproxy:

req.set\_proxy(gproxy, 'http')

response = openner.open(req)

if returnres: return response

data=response.read()

#print len(data)

return data

except:

print 'Error in getUrl'

traceback.print\_exc()

return None

def download\_chunks(URL, chunk\_size=4096, enc=None):

#conn=urllib2.urlopen(URL)

#print 'starting download'

conn=getUrl(URL,returnres=True,stream=True)

#while 1:

chunk\_size=chunk\_size\*100

for chunk in conn.iter\_content(chunk\_size=chunk\_size):

yield chunk

conn.close()

def download\_file(URL):

return ''.join(download\_chunks(URL))

def validate\_m3u(conn):

''' make sure file is an m3u, and returns the encoding to use. '''

return 'utf8'

mime = conn.headers.get('Content-Type', '').split(';')[0].lower()

if mime == 'application/vnd.apple.mpegurl':

enc = 'utf8'

elif mime == 'audio/mpegurl':

enc = 'iso-8859-1'

elif conn.url.endswith('.m3u8'):

enc = 'utf8'

elif conn.url.endswith('.m3u'):

enc = 'iso-8859-1'

else:

raise Exception("Stream MIME type or file extension not recognized")

if conn.readline().rstrip('\r\n') != '#EXTM3U':

raise Exception("Stream is not in M3U format")

return enc

def gen\_m3u(url, skip\_comments=True):

global cookieJar

conn = getUrl(url,returnres=True )#urllib2.urlopen(url)

redirurl=None

if conn.history:

print 'history'

redirurl=conn.url

enc = validate\_m3u(conn)

#print conn

if redirurl: yield 'f4mredirect:'+redirurl

for line in conn.iter\_lines():#.split('\n'):

line = line.rstrip('\r\n').decode(enc)

if not line:

# blank line

continue

elif line.startswith('#EXT'):

# tag

yield line

elif line.startswith('#'):

# comment

if skip\_comments:

continue

else:

yield line

else:

# media file

yield line

def parse\_m3u\_tag(line):

if ':' not in line:

return line, []

tag, attribstr = line.split(':', 1)

attribs = []

last = 0

quote = False

for i,c in enumerate(attribstr+','):

if c == '"':

quote = not quote

if quote:

continue

if c == ',':

attribs.append(attribstr[last:i])

last = i+1

return tag, attribs

def parse\_kv(attribs, known\_keys=None):

d = {}

for item in attribs:

k, v = item.split('=', 1)

k=k.strip()

v=v.strip().strip('"')

if known\_keys is not None and k not in known\_keys:

raise ValueError("unknown attribute %s"%k)

d[k] = v

return d

def handle\_basic\_m3u(url):

global iv

global key

global USEDec

global gauth

import urlparse

global callbackDRM

seq = 1

enc = None

nextlen = 5

duration = 5

targetduration=5

aesdone=False

redirurl=url

vod=False

for line in gen\_m3u(url):

if line.startswith('f4mredirect:'):

redirurl=line.split('f4mredirect:')[1]

continue

if line.startswith('#EXT'):

tag, attribs = parse\_m3u\_tag(line)

if tag == '#EXTINF':

duration = float(attribs[0])

elif tag == '#EXT-X-TARGETDURATION':

assert len(attribs) == 1, "too many attribs in EXT-X-TARGETDURATION"

targetduration = int(attribs[0])

pass

elif tag == '#EXT-X-MEDIA-SEQUENCE':

assert len(attribs) == 1, "too many attribs in EXT-X-MEDIA-SEQUENCE"

seq = int(attribs[0])

elif tag == '#EXT-X-KEY':

attribs = parse\_kv(attribs, ('METHOD', 'URI', 'IV'))

assert 'METHOD' in attribs, 'expected METHOD in EXT-X-KEY'

if attribs['METHOD'] == 'NONE':

assert 'URI' not in attribs, 'EXT-X-KEY: METHOD=NONE, but URI found'

assert 'IV' not in attribs, 'EXT-X-KEY: METHOD=NONE, but IV found'

enc = Nonee

elif attribs['METHOD'] == 'AES-128':

if not aesdone:

#aesdone=False there can be multple aes per file

assert 'URI' in attribs, 'EXT-X-KEY: METHOD=AES-128, but no URI found'

#from Crypto.Cipher import AES

codeurl=attribs['URI'].strip('"')

if gauth:

currentaesUrl=codeurl

codeurl=gauth

if codeurl.startswith("LSHex$"):

codeurl=codeurl.split('LSHex$')[1].decode("hex")

print 'code is ',codeurl.encode("hex")

if codeurl.startswith("LSDRMCallBack$"):

codeurlpath=codeurl.split('LSDRMCallBack$')[1]

codeurl='LSDRMCallBack$'+currentaesUrl

if codeurlpath and len(codeurlpath)>0 and callbackDRM==None:

print 'callback',codeurlpath

import importlib, os

foldername=os.path.sep.join(codeurlpath.split(os.path.sep)[:-1])

urlnew=''

if foldername not in sys.path:

sys.path.append(foldername)

try:

callbackfilename= codeurlpath.split(os.path.sep)[-1].split('.')[0]

callbackDRM = importlib.import\_module(callbackfilename)

print 'LSDRMCallBack imported'

except:

traceback.print\_exc()

elif not codeurl.startswith('http'):

import urlparse

codeurl=urlparse.urljoin(url, codeurl)

#key = download\_file(codeurl)

elif not codeurl.startswith('http'):

import urlparse

codeurl=urlparse.urljoin(url, codeurl)

#assert len(key) == 16, 'EXT-X-KEY: downloaded key file has bad length'

if 'IV' in attribs:

assert attribs['IV'].lower().startswith('0x'), 'EXT-X-KEY: IV attribute has bad format'

iv = attribs['IV'][2:].zfill(32).decode('hex')

assert len(iv) == 16, 'EXT-X-KEY: IV attribute has bad length'

else:

iv = '\0'\*8 + struct.pack('>Q', seq)

enc=(codeurl,iv)

#if not USEDec==3:

# enc = AES.new(key, AES.MODE\_CBC, iv)

#else:

# ivb=array.array('B',iv)

# keyb= array.array('B',key)

# enc=python\_aes.new(keyb, 2, ivb)

#enc = AES\_CBC(key)

#print key

#print iv

#enc=AESDecrypter.new(key, 2, iv)

else:

assert False, 'EXT-X-KEY: METHOD=%s unknown'%attribs['METHOD']

elif tag == '#EXT-X-PROGRAM-DATE-TIME':

assert len(attribs) == 1, "too many attribs in EXT-X-PROGRAM-DATE-TIME"

# TODO parse attribs[0] as ISO8601 date/time

pass

elif tag == '#EXT-X-ALLOW-CACHE':

# XXX deliberately ignore

pass

elif tag == 'EXT-X-PLAYLIST-TYPE:VOD':

vod=True

pass

#EXT-X-PLAYLIST-TYPE:VOD

elif tag == '#EXT-X-ENDLIST':

assert not attribs

yield None

return

elif tag == '#EXT-X-STREAM-INF':

raise ValueError("don't know how to handle EXT-X-STREAM-INF in basic playlist")

elif tag == '#EXT-X-DISCONTINUITY':

assert not attribs

print "[warn] discontinuity in stream"

elif tag == '#EXT-X-VERSION':

assert len(attribs) == 1

if int(attribs[0]) > SUPPORTED\_VERSION:

print "[warn] file version %s exceeds supported version %d; some things might be broken"%(attribs[0], SUPPORTED\_VERSION)

#else:

# raise ValueError("tag %s not known"%tag)

else:

if not line.startswith('http'):

line=urlparse.urljoin(redirurl, line)

yield (seq, enc, duration, targetduration, line ,vod)

seq += 1

def player\_pipe(queue, control,file):

while 1:

block = queue.get(block=True)

if block is None: return

file.write(block)

file.flush()

def send\_back(data,file):

file.write(data)

file.flush()

def downloadInternal(url,file,maxbitrate=0,stopEvent=None , callbackpath="",callbackparam="", testing=False):

global key

global iv

global USEDec

global cookieJar

global clientHeader

global nsplayer

global callbackDRM

if stopEvent and stopEvent.isSet():

return False

dumpfile = None

#dumpfile=open('c:\\temp\\myfile.mp4',"wb")

variants = []

variant = None

veryfirst=True

#url check if requires redirect

redirurl=url

utltext=''

try:

print 'going for url ',url

res=getUrl(url,returnres=True )

print 'here ', res

if res.history:

print 'history is',res.history

redirurl=res.url

url=redirurl

utltext=res.text

res.close()

if testing: return True

except: traceback.print\_exc()

print 'redirurl',redirurl

if 'EXT-X-STREAM-INF' in utltext:

try:

for line in gen\_m3u(redirurl):

if line.startswith('#EXT'):

tag, attribs = parse\_m3u\_tag(line)

if tag == '#EXT-X-STREAM-INF':

variant = attribs

elif variant:

variants.append((line, variant))

variant = None

print 'variants',variants

if len(variants)==0: url=redirurl

if len(variants) == 1:

url = urlparse.urljoin(redirurl, variants[0][0])

elif len(variants) >= 2:

print "More than one variant of the stream was provided."

choice=-1

lastbitrate=0

print 'maxbitrate',maxbitrate

for i, (vurl, vattrs) in enumerate(variants):

print i, vurl,

for attr in vattrs:

key, value = attr.split('=')

key = key.strip()

value = value.strip().strip('"')

if key == 'BANDWIDTH':

print 'bitrate %.2f kbps'%(int(value)/1024.0)

if int(value)<=int(maxbitrate) and int(value)>lastbitrate:

choice=i

lastbitrate=int(value)

elif key == 'PROGRAM-ID':

print 'program %s'%value,

elif key == 'CODECS':

print 'codec %s'%value,

elif key == 'RESOLUTION':

print 'resolution %s'%value,

else:

print "unknown STREAM-INF attribute %s"%key

#raise ValueError("unknown STREAM-INF attribute %s"%key)

print

if choice==-1: choice=0

#choice = int(raw\_input("Selection? "))

print 'choose %d'%choice

url = urlparse.urljoin(redirurl, variants[choice][0])

except:

raise

print 'final url',url

last\_seq = -1

targetduration = 5

changed = 0

fails=0

maxfails=5

nsplayer=False

print 'inside HLS RETRY'

try:

#file.write(b'FLV\x01')

#file.write(b'\x01')

#file.write(b'\x00\x00\x00\x09')

# FLV File body

#file.write(b'\x00\x00\x00\x09')

while 1==1:#thread.isAlive():

reconnect=False

vod=False

if fails>maxfails:

#stopEvent.set()

break

if stopEvent and stopEvent.isSet():

return False

try:

medialist = list(handle\_basic\_m3u(url))

if len(medialist)==0: raise Exception('empty m3u8')

print medialist

if testing: return True

except Exception as inst:

print 'here in exp',inst

print fails

fails+=1

if testing and fails>6: return False

if testing==False and '403' in repr(inst).lower() and callbackpath and len(callbackpath)>0:

print 'callback'

import importlib, os

foldername=os.path.sep.join(callbackpath.split(os.path.sep)[:-1])

urlnew=''

if foldername not in sys.path:

sys.path.append(foldername)

try:

callbackfilename= callbackpath.split(os.path.sep)[-1].split('.')[0]

callbackmodule = importlib.import\_module(callbackfilename)

urlnew,cjnew=callbackmodule.f4mcallback(callbackparam, 1, inst, cookieJar , url, clientHeader)

except: traceback.print\_exc()

if urlnew and len(urlnew)>0 and urlnew.startswith('http'):

print 'got new url',url

url=urlnew

cookieJar= cjnew

continue

else:

return

if '403' in repr(inst).lower() or '401' in repr(inst).lower():

if fails in [1,4,5,10,15,19]:

nsplayer=True

else:

nsplayer=False

print 'nsplayer',nsplayer

xbmc.sleep(1000)

continue

nsplayer=False

playedSomething=False

if medialist==None: return

## choose to start playback three files from the end, since this is a live stream

#medialist = medialist[-6:]

#print 'medialist',medialist

addsomewait=False

lastKeyUrl=""

lastkey=None

playedduration=0

st=time.time()

for media in medialist:

if stopEvent and stopEvent.isSet():

return False

if media is None:

#send\_back('G'+chr(254)+chr(255)+('\0'\*1), file)

#queue.put(None, block=True)

if stopEvent:

print 'set events'

stopEvent.set()

return False

seq, encobj, duration, targetduration, media\_url,vod = media

if seq > last\_seq:

#print 'downloading.............',url

enc=None

if encobj:

codeurl,iv=encobj

if codeurl<>lastKeyUrl:

if codeurl.startswith('http'):

key = download\_file(codeurl)

elif codeurl.startswith('LSDRMCallBack$'):

key=callbackDRM.DRMCallback(codeurl.split('LSDRMCallBack$')[1],url)

else:

key = codeurl

codeurl=lastKeyUrl

else:

key=lastkey

lastkey=key

if not USEDec==3:

enc = AES.new(key, AES.MODE\_CBC, iv)

else:

ivb=array.array('B',iv)

keyb= array.array('B',key)

enc=python\_aes.new(keyb, 2, ivb)

#enc=AESDecrypter.new(key, 2, iv)

try:

data=None

try:

print 'downloading', urlparse.urljoin(url, media\_url)

#for chunk in download\_chunks(urlparse.urljoin(url, media\_url)):

for chunk in download\_chunks(media\_url):

if stopEvent and stopEvent.isSet():

return False

print 'sending chunk', len(chunk)

if enc:

if not USEDec==3:

chunk = enc.decrypt(chunk)

else:

chunkb=array.array('B',chunk)

chunk = enc.decrypt(chunkb)

chunk="".join(map(chr, chunk))

send\_back(chunk,file)

data="send"

playedduration+=duration

addsomewait=True

except Exception as inst:

print 'xxxx',repr(inst)

if 'forcibly closed' in repr(inst):

print 'returning'

return False

if stopEvent and stopEvent.isSet():

return False

if data and len(data)>0:# chunk in download\_chunks(urlparse.urljoin(url, media\_url),enc=encobj):

#if not veryfirst:

# if dumpfile: dumpfile.write(chunk)

# #queue.put(chunk, block=True)

# send\_back(data,file)

# #print '3. chunk available %d'%len(chunk)

veryfirst=False

last\_seq = seq

changed = 1

playedSomething=True

fails=0

maxfails=20

else:

reconnect=True

fails+=1

break

except: pass

if vod: return True

if playedSomething == 1:

# initial minimum reload delay

timetowait=int(targetduration - (time.time()-st))#

if (timetowait)>0:

print 'sleeping because targetduration',timetowait

for t in range(0,timetowait):

xbmc.sleep(1000)

print 'sleeep for 1sec',t

if stopEvent and stopEvent.isSet():

return False

'''elif changed == 0:

# first attempt

time.sleep(targetduration\*0.5)

elif changed == -1:

# second attempt

time.sleep(targetduration\*1.5)

else:

# third attempt and beyond

time.sleep(targetduration\*3.0)

changed -= 1

'''

if not playedSomething:

xbmc.sleep(3000+ (3000 if addsomewait else 0))

except:

raise