daniellivingston Add Python script for ... Latest commit c38e8b0 yesterday History

1 contributor

407 lines (309 sloc) | 11.4 KB Raw Blame

```
#!/usr/bin/env python

"""
Manipulate PSARC archives used by Rocksmith 2014.

Requires pycrypto (pip install pycrypto).
pip install pycryptodome

Usage:
    psarc.py pack DIRECTORY...
    psarc.py unpack FILE...
    psarc.py convert FILE...
"""

from Crypto.Cipher import AES
from Crypto.Util import Counter

import struct
import zlib
import os
import hashlib
import sys
import json

import codecs

import sys
```

```
print("psarc.py running on Python version %s.%s" % (sys.version_info.major,sys.v

MAGIC = "PSAR"
VERSION = 65540
COMPRESSION = "zlib"
ARCHIVE_FLAGS = 4
ENTRY_SIZE = 30
BLOCK_SIZE = 65536

ARC_KEY = 'C53DB23870A1A2F71CAE64061FDD0E1157309DC85204D4C5BFDF25090DF2572C'
ARC_IV = 'E915AA018FEF71FC508132E4BB4CEB42'

MAC_KEY = '9821330E34B91F70D0A48CBD625993126970CEA09192C0E6CDA676CC9838289D'
PC_KEY = 'CB648DF3D12A16BF71701414E69619EC171CCA5D2A142E3E59DE7ADDA18A3A30'

PRF_KEY = '728B369E24ED0134768511021812AFC0A3C25D02065F166B4BCC58CD2644F29E'

CONFIG_KEY = '378B9026EE7DE70B8AF124C1E30978670F9EC8FD5E7285A86442DD73068C0473'


def pad(data, blocksize=16):
    """Zeros padding"""
    #So we need zeroes in order to match AES's encoding scheme which breaks
    #the data into 16-byte chunks. If we have 52 bytes to decode, then we have
    #3 full 16-byte blocks, and 4 left over. This means we need 12 bytes worth
    #of padding. That means we get 12 bytes worth of zeroes.
    padding = (blocksize - len(data)) % blocksize
    #Modify this to return a bytes object, since that's what pycrypto needs.
    return data + bytes(padding)


def path2dict(path):
    """Reads a path into a dictionary"""
    output = {}
    for dirpath, _, filenames in os.walk(path):
        for filename in filenames:
            fullpath = os.path.join(dirpath, filename)
            name = fullpath[len(path) + 1:]

            with open(fullpath, 'rb') as fstream:
                output[name] = fstream.read()

    return output


```

```
def decrypt_profile(stream):
    """For *_prfldb, crd and profile.json files"""
    s = stream.read()
    size = struct.unpack('<L', s[16:20])[0]

    cipher = AES.new(codecs.decode(PRF_KEY,'hex'))
    x = zlib.decompress(cipher.decrypt(pad(s[20:])))
    assert(size == len(x))

    return json.loads(x[:-1])  # it's a long C string


def stdout_same_line(line):
    """Prepend carriage return and output to stdout"""
    sys.stdout.write('\r' + line[:80])
    sys.stdout.flush()


def aes_ctr(data, key, ivector, encrypt=True):
    """AES CTR Mode"""
    output = bytes()
    #First param: Number of bits of the cipher. 64 is not meaningful.
    ctr = Counter.new(64, initial_value = ivector)
    cipher = AES.new(codecs.decode(key,'hex'), mode=AES.MODE_CTR, counter=ctr)

    if encrypt:
        output += cipher.encrypt(pad(data))
    else:
        output += cipher.decrypt(pad(data))

    return output


def decrypt_sng(data, key):
    """Decrypt SNG. Data consist of a 8 bytes header, 16 bytes initialization
    vector and payload and the DSA signature. Payload is first decrypted using
    AES CTR and then zlib decompressed. Size is checked."""
    #Changed conversion from bytes to int for initial value.
    decrypted = aes_ctr(data[24:], key, int(codecs.encode(data[8:24],'hex'),16),
    length = struct.unpack('<L', decrypted[:4])[0]  # file size
    payload = ''
    try:
        payload = zlib.decompress(decrypted[4:])
        assert len(payload) == length
    except Exception as e:
```

```
118             #print('An error occurred while processing sng!')
119             #print(e)
120             payload = decrypted
121 
122     return payload
123 
124 
125 def encrypt_sng(data, key):
126     """Encrypt SNG"""
127     output = struct.pack('<LL', 0x4a, 3)  # the header
128 
129     payload = struct.pack('<L', len(data))
130     payload += zlib.compress(data, zlib.Z_BEST_COMPRESSION)
131 
132     ivector = bytes(16)
133     output += ivector
134     output += aes_ctr(payload, key, ivector)
135     return output + bytes(56)
136 
137 
138 def decrypt_config(data):
139     """For pkgconfig.ini"""
140     data = data[:-56]  # remove signature
141 
142     cipher = AES.new(codecs.decode(CONFIG_KEY,'hex'))
143     t = cipher.decrypt(data)
144     if t.find('\x00') > -1:  # padding was applied
145         return t[:t.index('\x00')]
146     return t
147 
148 
149 def encrypt_config(data):
150     """For pkgconfig.ini"""
151     cipher = AES.new(codecs.decode(CONFIG_KEY,'hex'))
152     t = cipher.encrypt(pad(data))
153     t += bytes(56)
154     return t
155 
156 
157 def read_entry(filestream, entry):
158     """Extract zlib for one entry"""
159     data = bytes()
160 
161     length = entry['length']
162     zlength = entry['zlength']
```

```
    filestream.seek(entry['offset'])

    i = 0
    while len(data) < length:
        if zlength[i] == 0:
            data += filestream.read(BLOCK_SIZE)
        else:
            chunk = filestream.read(zlength[i])
            try:
                data += zlib.decompress(chunk)
            except zlib.error:
                data += chunk
        i += 1

    # Post process for sng
    if entry['filepath'].find('songs/bin/macos/') > -1:
        data = decrypt_sng(data, MAC_KEY)
    elif entry['filepath'].find('songs/bin/generic/') > -1:
        data = decrypt_sng(data, PC_KEY)

    # Requires bypass for ini
    # if entry['filepath'] == 'pkgconfig.ini':
    #     data = decrypt_config(data)

    return data


def create_entry(name, data):
    """Chunk a file"""

    # Pre process for sng
    if name.find('songs/bin/macos/') > -1:
        data = encrypt_sng(data, MAC_KEY)
    elif name.find('songs/bin/generic/') > -1:
        data = encrypt_sng(data, PC_KEY)

    # Requires bypass for ini
    # if name == 'pkgconfig.ini':
    #     data = encrypt_config(data)

    zlength = []
    output = ''

    i = 0
    while i < len(data):
```

```
        raw = data[i:i + BLOCK_SIZE]
        i += BLOCK_SIZE

        compressed = zlib.compress(raw, zlib.Z_BEST_COMPRESSION)
        if len(compressed) < len(raw):
            output += compressed
            zlength.append(len(compressed))
        else:
            output += raw
            zlength.append(len(raw) % BLOCK_SIZE)

    return {
        'filepath': name,
        'zlength': zlength,
        'length': len(data),
        'data': output,
        'md5': md5.new(name).digest() if name != '' else bytes(16)
    }


def cipher_toc():
    """AES CFB Mode"""
    return AES.new(codecs.decode(ARC_KEY,'hex'), mode=AES.MODE_CFB,
                   IV=codecs.decode(ARC_IV,'hex'), segment_size=128)


def read_toc(filestream):
    """Read entry list and Z-fragments.
    Returns a list of entries to be used with read_entry."""

    entries = []
    zlength = []

    filestream.seek(0)
    header = struct.unpack('>4sL4sLLLLL', filestream.read(32))

    toc_size = header[3] - 32
    n_entries = header[5]
    toc = cipher_toc().decrypt(pad(filestream.read(toc_size)))
    toc_position = 0

    idx = 0
    while idx < n_entries:
        data = toc[toc_position:toc_position + ENTRY_SIZE]

```

```
        entries.append({
            'md5': data[:16],
            'zindex': struct.unpack('>L', data[16:20])[0],
            'length': struct.unpack('>Q', b'\x00'*3 + data[20:25])[0],
            'offset': struct.unpack('>Q', b'\x00'*3 + data[25:])[0]
        })
        toc_position += ENTRY_SIZE
        idx += 1

    idx = 0
    while idx < (toc_size - ENTRY_SIZE * n_entries) / 2:
        data = toc[toc_position:toc_position + 2]
        zlength.append(struct.unpack('>H', data)[0])
        toc_position += 2
        idx += 1

    for entry in entries:
        entry['zlength'] = zlength[entry['zindex']:]

    # Process the first entry as it contains the file listing
    entries[0]['filepath'] = ''
    filepaths = read_entry(filestream, entries[0]).split()
    for entry, filepath in zip(entries[1:], filepaths):
        entry['filepath'] = filepath.decode("utf-8")

    return entries[1:]


def create_toc(entries):
    """Build an encrypted TOC for a given list of entries."""

    offset = 0
    zindex = 0
    zlength = []
    for entry in entries:
        entry['offset'] = offset
        offset += len(entry['data'])

        entry['zindex'] = zindex
        zindex += len(entry['zlength'])

        zlength += entry['zlength']

    toc_size = 32 + ENTRY_SIZE * len(entries) + 2 * len(zlength)

```

```
    header = struct.pack('>4sL4sLLLLL', MAGIC, VERSION, COMPRESSION,
                         toc_size, ENTRY_SIZE, len(entries),
                         BLOCK_SIZE, ARCHIVE_FLAGS)

    toc = ''
    for entry in entries:
        toc += entry['md5']
        toc += struct.pack('>L', entry['zindex'])
        toc += struct.pack('>Q', entry['length'])[-5:]
        toc += struct.pack('>Q', entry['offset'] + toc_size)[-5:]

    for i in zlength:
        toc += struct.pack('>H', i)

    # the [:toc_size] seems a little odd, but padding is not applied
    # in official PSARC either
    return (header + cipher_toc().encrypt(pad(toc)))[:toc_size]


def extract_psarc(filename):
    """Extract a PSARC to disk"""
    basepath = os.path.basename(filename)[:-6]

    with open(filename, 'rb') as psarc:
        entries = read_toc(psarc)
        logmsg = 'Extracting ' + basepath + ' {0}/' + str(len(entries))

        for idx, entry in enumerate(entries):
            stdout_same_line(logmsg.format(idx + 1))
            fname = os.path.join(basepath, entry['filepath'])
            data = read_entry(psarc, entry)
            path = os.path.dirname(fname)
            if not os.path.exists(path):
                os.makedirs(path)
            with open(fname, 'wb') as fstream:
                fstream.write(data)


def create_psarc(files, filename):
    """Writes a dictionary filepath -> data to a PSARC file"""
    # Order is reversed
    filenames = reversed(sorted(files.keys()))
    entries = [create_entry('', '\n'.join(filenames))]

    logmsg = 'Creating ' + filename + ' {0}/' + str(len(files))
```

```
    for idx, (name, data) in enumerate(reversed(sorted(files.items()))):
        stdout_same_line(logmsg.format(idx + 1))
        entries.append(create_entry(name, data))

    with open(filename, 'wb') as fstream:
        fstream.write(create_toc(entries))
        for entry in entries:
            fstream.write(entry['data'])


def change_path(data, osx2pc):
    """Changing path"""
    if osx2pc:
        data = data.replace('audio/mac', 'audio/windows')
        data = data.replace('bin/macos', 'bin/generic')
    else:
        data = data.replace('audio/windows', 'audio/mac')
        data = data.replace('bin/generic', 'bin/macos')
    return data


# TODO logic for converting needs cleanup
def convert(filename):
    """Convert between PC and Mac PSARC"""
    content = {}

    osx2pc = False
    outname = filename
    if filename.endswith('_m.psarc'):
        outname = filename.replace('_m.psarc', '_p.psarc')
        osx2pc = True
    else:
        outname = filename.replace('_p.psarc', '_m.psarc')

    with open(filename, 'rb') as psarc:
        entries = read_toc(psarc)
        for entry in entries:
            data = read_entry(psarc, entry)

            if entry['filepath'].endswith('aggregategraph.nt'):
                data = change_path(data, osx2pc)
                if osx2pc:
                    data = data.replace('macos', 'dx9')
                else:
                    data = data.replace('dx9', 'macos')
```

```
388
389             content[change_path(entry['filepath'], osx2pc)] = data
390
391     create_psarc(content, outname)
392
393
394 if __name__ == '__main__':
395     from docopt import docopt
396     args = docopt(__doc__)
397
398     if args['unpack']:
399         for f in args['FILE']:
400             extract_psarc(f)
401     elif args['pack']:
402         for d in args['DIRECTORY']:
403             d = os.path.normpath(d)
404             create_psarc(path2dict(d), d + '.psarc')
405     elif args['convert']:
406         for f in args['FILE']:
407             convert(f)
```