Tiny and fast ASN.1 decoder in Python

8.11.2014 Version 1.0

Jens Getreu

Table of Contents

`asn1tinydecoder.py` is a simple and fast ASN.1 decoder without external libraries designed to parse large files.

There is a good "Pyasn1" ASN.1 python library out there. It is very complete, but learning how to use it is quite time consuming. Furthermore Pyasn1 is not designed to parse large files. This why I wrote this tiny ASN.1 decoder. It's design goal was to be as fast as possible (with Python).

A practical application can be found in Search serial in large certificate revocation lists. This program creates an index allowing fast search in big ASN.1 data structures.

1. The ASN1 decoder

1.1. Decoder API

The decoder API consists of 7 functions documented below.

ASN1 node

In `asn1tinydecoder.py` a "node" is a pointer to an ASN.1 chunk of Bytes containing 3 parts:

```
[type Byte, length Bytes, value Bytes]
```

A node represented by a Python tuple:

```
(ixs, ixf, ixl)
```

`ixs`
: Points to the type Byte which is the first Byte of the chunk.

`ixf`
: Points to the first value Byte.

`ixl`
: Points to the last value Byte which is the last Byte of the chunk.

Navigate

In order to browse through ASN.1 tree structures you need only 3 functions to navigate:

All functions of `asn1tinydecoder.py` are stateless.

`asn1_node_root()`
: Points to the first node in the tree structure. This is the root node.

`asn1_node_next()`
: Skips and points to the next node.

`asn1_node_first_child()`
: Opens a ASN1 *sequence* or *set* and points to the first node inside the *sequence* or *set*.

`asn1_node_is_child_of()`
: Controls loops over lists i.e. *sequence* or *set*. It returns `true` if one ASN1 chunk is inside another chunk. [1: The function tests if the two nodes are in a parent-child or grand parent-child relation. The order of the two parameters does not matter.] See example in function `extract_crl_info()` in ASN1 decoder usage example.

Accessing node's data

Once you found the right node you can access the node's data with:

`asn1_get_value()`
: Gets the bytestring of value Bytes of the pointed node.

There is no check if the node contains the data type you expect. If you can please use `asn1_get_value_of_type()` instead.

`asn1_get_value_of_type()`
: Same as the above, but first checks if the pointed node is of a given type. Recognized types are: `BOOLEAN`, `INTEGER`, `BIT STRING`, `OCTET STRING`, `NULL`, `OBJECT IDENTIFIER`, `SEQUENCE`, `SET`, `PrintableString`, `IA5String`, `UTCTime`, `ENUMERATED`, `UTF8String`, `PrintableString`.

`asn1_get_all()`
: Gets the bytes of the whole node i.e. type Byte, length Bytes and value Bytes in one string.

All data is returned in raw format exactly as it is stored in the ASN.1 chunk. You need to convert it yourself. [3: The most commonly used converters `bitstr_to_bytestr()` and `bytestr_to_int` are provided by `asn1tinydecoder.py`.]

1.2. Decoder source code

Download `asn1tinydecoder.py` here.

ASN1 decoder source code

```
####################### BEGIN ASN1 DECODER ############################

# Author: Jens Getreu, 8.11.2014
```

```
##### NAVIGATE

# The following 4 functions are all you need to parse an ASN1 structure

# gets the first ASN1 structure in der
def asn1_node_root(der):
    return asn1_read_length(der,0)

# gets the next ASN1 structure following (ixs,ixf,ixl)
def asn1_node_next(der, (ixs,ixf,ixl)):
    return asn1_read_length(der,ixl+1)

# opens the container (ixs,ixf,ixl) and returns the first ASN1 inside
def asn1_node_first_child(der, (ixs,ixf,ixl)):
    if ord(der[ixs]) & 0x20 != 0x20:
        raise ValueError('Error: can only open constructed types. '
                +'Found type: 0x'+der[ixs].encode("hex"))
    return asn1_read_length(der,ixf)

# is true if one ASN1 chunk is inside another chunk.
def asn1_node_is_child_of((ixs,ixf,ixl), (jxs,jxf,jxl)):
    return ( (ixf <= jxs ) and (jxl <= ixl) ) or \
           ( (jxf <= ixs ) and (ixl <= jxl) )

##### END NAVIGATE



##### ACCESS PRIMITIVES

# get content and verify type byte
def asn1_get_value_of_type(der,(ixs,ixf,ixl),asn1_type):
    asn1_type_table = {
    'BOOLEAN':           0x01,  'INTEGER':           0x02,
    'BIT STRING':        0x03,  'OCTET STRING':      0x04,
    'NULL':              0x05,  'OBJECT IDENTIFIER': 0x06,
    'SEQUENCE':          0x70,  'SET':               0x71,
    'PrintableString':   0x13,  'IA5String':         0x16,
    'UTCTime':           0x17,  'ENUMERATED':        0x0A,
    'UTF8String':        0x0C,  'PrintableString':   0x13,
    }
    if asn1_type_table[asn1_type] != ord(der[ixs]):
        raise ValueError('Error: Expected type was: '+
            hex(asn1_type_table[asn1_type])+
            ' Found: 0x'+der[ixs].encode('hex'))
    return der[ixf:ixl+1]

# get value
```

```
def asn1_get_value(der,(ixs,ixf,ixl)):
    return der[ixf:ixl+1]

# get type+length+value
def asn1_get_all(der,(ixs,ixf,ixl)):
    return der[ixs:ixl+1]

##### END ACCESS PRIMITIVES



##### HELPER FUNCTIONS

# converter
def bitstr_to_bytestr(bitstr):
    if bitstr[0] != '\x00':
        raise ValueError('Error: only 00 padded bitstr can be converted to bytestr!')
    return bitstr[1:]

# converter
def bytestr_to_int(s):
    # converts bytestring to integer
    i = 0
    for char in s:
        i <<= 8
        i |= ord(char)
    return i

# ix points to the first byte of the asn1 structure
# Returns first byte pointer, first content byte pointer and last.
def asn1_read_length(der,ix):
    first= ord(der[ix+1])
    if  (ord(der[ix+1]) & 0x80) == 0:
        length = first
        ix_first_content_byte = ix+2
        ix_last_content_byte = ix_first_content_byte + length -1
    else:
        lengthbytes = first & 0x7F
        length = bytestr_to_int(der[ix+2:ix+2+lengthbytes])
        ix_first_content_byte = ix+2+lengthbytes
        ix_last_content_byte = ix_first_content_byte + length -1
    return (ix,ix_first_content_byte,ix_last_content_byte)

##### END HELPER FUNCTIONS


####################### END ASN1 DECODER ############################
```

2. ASN1 decoder usage example

CRL lists can be hundreds of MB long. The (otherwise very good) ASN1 python library takes hours to run through such large structures. `asn1tinydecoder` does the same in some seconds.

See the source code below for examples how the decoder can be used.

2.1. Search serial in large certificate revocation lists

`search_serial_in_large_CRL.py` searches serials in large CRLs.

It runs through large ASN1 structures creating an index dictionary `serials_idx`. There it saves certificate serials and pointers to corresponding certificate data chunks (see function `extract_crl_info()`).

The dictionary is then used to determine if a given serial is revoked or not. This index allows fast search.

The function `search_certificate()` prints out the corresponding revoked certificate data.

Performance

Indexing a 19MB CRL with 266616 certificates takes less then 6 seconds on my netbook with *AMD E-450* CPU.

2.1.1. Source code

```
################# BEGIN ASN1 DECODER USAGE EXAMPLE  ###################

# Author: Jens Getreu, 8.11.2014, Version 1.0


# Please install also dumpasn1 package with you package manager!
from asn1tinydecoder import asn1_node_root, asn1_get_all, asn1_get_value, \
                        asn1_get_value_of_type, asn1_node_next, asn1_node_first_child, \
                        asn1_read_length, asn1_node_is_child_of, \
                        bytestr_to_int, bitstr_to_bytestr

import hashlib, datetime
from subprocess import Popen, PIPE, STDOUT # for debugging only

# This code illustrates the usage of asn1decoder

# Install dumpasn1 package to make this work.
# For debugging only, prints ASN1 structures nicely.
def dump_asn1(der):
    p = Popen(['dumpasn1','-a', '-'], stdout=PIPE, stdin=PIPE,
            stderr=STDOUT)
    dump = p.communicate(input=der)[0]
    return dump



# This function extracts some header fields of the CRL list
# and stores pointers to the list entries in a dictionary

def extract_crl_info(crl_der):
    # unpack sequence
    i = asn1_node_root(crl_der)
    # unpack sequence
    i = asn1_node_first_child(crl_der,i)
    crl_signed_content= i

    # get 1. item inside (version)
    i = asn1_node_first_child(crl_der,i)
    # advance 1 item (Algoidentifier)
    i = asn1_node_next(crl_der,i)
    # advance 1 item (email, CN etc.)
    i = asn1_node_next(crl_der,i)
    # advance 1 item
    i = asn1_node_next(crl_der,i)
    bytestr = asn1_get_value_of_type(crl_der,i,'UTCTime')
    crl_not_valid_before = datetime.datetime.strptime(bytestr,'%y%m%d%H%M%SZ')
    # advance 1 item
    i = asn1_node_next(crl_der,i)
    bytestr = asn1_get_value_of_type(crl_der,i,'UTCTime')
    crl_not_valid_after = datetime.datetime.strptime(bytestr,'%y%m%d%H%M%SZ')

    # advance 1 item (the list)
    i = asn1_node_next(crl_der,i)

    # Stores for every certificate entry the serial number and and
    # 3 pointers indication the position of the certificate entry.
    # Returns a dictionary.
    # key = certificate serial number
    # value = 3 pointers to certificate entry in CRL

    #open and read 1. item
    j = asn1_node_first_child(crl_der,i)
    serials_idx = {}

    while asn1_node_is_child_of(i,j):
        #read 1. interger inside item
        k = asn1_node_first_child(crl_der,j)
        serial = bytestr_to_int(
            asn1_get_value_of_type(crl_der,k,'INTEGER'))
        #store serial and the asn1 container position
        serials_idx[serial] = j

        # point on next item in the list
        j = asn1_node_next(crl_der,j)

    # advance 1 item
    i = asn1_node_next(crl_der,i)
    # advance 1 item (obj. identifier)
    i = asn1_node_next(crl_der,i)
    # advance 1 item (signature)
    i = asn1_node_next(crl_der,i)
    # content is crl_signature
    crl_signature = bitstr_to_bytestr(
        asn1_get_value_of_type(crl_der,i,'BIT STRING'))

    return crl_not_valid_before, crl_not_valid_after, \
            crl_signature, \
            crl_signed_content,serials_idx



# Print the header fields and the dictionary
def search_certificate(crl_der,serial,(a,b,c,d,serials_idx)):
    print '*** Some information about the CRL'
    print 'crl_not_valid_before: ',a
    print 'crl_not_valid_after:  ',b
    print 'crl_signature:        ', \
            c.encode('hex')[:30],'... ',len(c),' Bytes'
    (ixs,ixf,ixl) = d
    print 'crl_signed_content:   ', d, ixl+1 - ixs,'Bytes'
    #print dump_asn1(d)
    print
    print '*** The CRL lists', len(serials_idx),'certificates.'
    if len(serials_idx) <= 10 :
        for c,p in serials_idx.items() :
            print 'serial: ',c,'  position:',p
    print

    print '*** Search in CRL for serial no:', serial
    print

    if serial in serials_idx:
        print '*** SERIAL FOUND IN LIST!:'
        print '**      Revoked certificat data'
        print '- Certificat serial no: ', serial
        # Now use the pointers to print the certificate entries.
        print '- Decoded ASN1 data:'
        p = serials_idx[serial]
        print dump_asn1(asn1_get_all(crl_der,p))
        print




### Main program
crl_filename = 'www.sk.ee-crl.crl'
search_serial = 1018438612

print "****** INDEXING CRL:", crl_filename
print
crl_der = open(crl_filename).read()
dictionary = extract_crl_info(crl_der)
search_certificate(crl_der,search_serial,dictionary)
#print crl_der.encode("hex")
#print dump_asn1(crl_der)
print
print

crl_filename = 'www.sk.ee-esteid2011.crl'
search_serial = 131917818486436565990004418739006228479

print "****** INDEXING CRL:", crl_filename
print
crl_der = open(crl_filename).read()
dictionary = extract_crl_info(crl_der)
search_certificate(crl_der,search_serial,dictionary)


################## END DECODER USAGE EXAMPLE  ######################
```

2.1.2. Program output

```
****** INDEXING CRL: www.sk.ee-crl.crl

*** Some information about the CRL
crl_not_valid_before:  2014-08-15 09:42:19
crl_not_valid_after:   2014-11-23 09:42:19
crl_signature:         4ea0be0063cffed880f5a1cafa3a5a ...  256  Bytes
crl_signed_content:    (4, 8, 511) 508 Bytes

*** The CRL lists 8 certificates.
```

```
serial:  999183360   position: (155, 157, 191)
serial:  1043934336   position: (266, 268, 302)
serial:  1084184741   position: (414, 416, 450)
serial:  1167825894   position: (192, 194, 228)
serial:  1018438612   position: (340, 342, 376)
serial:  1018438937   position: (377, 379, 413)
serial:  1018259643   position: (303, 305, 339)
serial:  1167830238   position: (229, 231, 265)

*** Search in CRL for serial no: 1018438612

*** SERIAL FOUND IN LIST!:
**      Revoked certificat data
- Certificat serial no:  1018438612
- Decoded ASN1 data:
  0  35: SEQUENCE {
  2   4:   INTEGER 1018438612
  8  13:   UTCTime 01/12/2009 10:10:04 GMT
 23  12:   SEQUENCE {
 25  10:     SEQUENCE {
 27   3:       OBJECT IDENTIFIER cRLReason (2 5 29 21)
 32   3:       OCTET STRING 0A 01 05
       :       }
       :     }
       :   }



****** INDEXING CRL: www.sk.ee-esteid2011.crl

*** Some information about the CRL
crl_not_valid_before:  2014-11-08 02:22:05
crl_not_valid_after:   2014-11-08 14:22:05
crl_signature:         a3659dab04a25d7e128b836fcbe844 ...  256  Bytes
crl_signed_content:    (6, 12, 19995446) 19995441 Bytes

*** The CRL lists 266616 certificates.

*** Search in CRL for serial no: 131917818486436565990004418739006228479

*** SERIAL FOUND IN LIST!:
**      Revoked certificat data
- Certificat serial no:  131917818486436565990004418739006228479
- Decoded ASN1 data:
  0  73: SEQUENCE {
  2  16:   INTEGER 63 3E 72 9B 4B BD B7 7F 51 24 F2 20 A8 AA 47 FF
 20  13:   UTCTime 18/03/2014 14:12:25 GMT
 35  38:   SEQUENCE {
```

```
 37  10:     SEQUENCE {
 39   3:       OBJECT IDENTIFIER cRLReason (2 5 29 21)
 44   3:       OCTET STRING 0A 01 04
       :       }
 49  24:     SEQUENCE {
 51   3:       OBJECT IDENTIFIER invalidityDate (2 5 29 24)
 56  17:       OCTET STRING 18 0F 32 30 31 34 30 33 31 38 31 34 31 32 32 35 5A
       :       }
       :     }
       :   }
```