ASN.1 Misuse

Carl M. Ellison *

September 15, 1995

1 Introduction

The ASN.1 standard (Abstract Syntax Notation 1) is proliferating, in spite of cries of anguish by computer professionals who are faced with implementing to those standards. The standard is clearly flawed.

Some of those flaws come from its complexity and ambiguity – a product most likely of having started down the wrong path and been subject to corrections by various interested parties along the way. It has acquired various warts and is apparently the result of a committee effort.

Other flaws come from its misuse, and those are the ones addressed in this paper.

2 Goals in conflict

ASN.1 is viewed differently by writers of standards and implementors. Neither group is unanimous in its evaluation, but it tends to be predominantly favored by the former and predominantly despised by the latter.

2.1 The writer of standards

With ASN.1 one can declare in a compact and readable way that a given data structure is to contain (for example) 2 fields, the first of which is an

*Trusted Information Systems, Inc.; 3060 Washington Road; Glenwood MD 21738; Email address: `cme@tis.com`

optional byte string and the second of which is an integer whose maximum size is yet to be determined. These structures and fields can be named with arbitrary length names, but the current practice is to use long names without abbreviations and with capital letters indicating starts of words within a name. forExample, thisWouldBeASampleName. As a result, names can be explicit enough (and long enough) that comments are not needed to explain the function and content of a defined structure.

ASN.1 has an additional redeeming feature to a person drafting a standard. It is not tied to any existing computer language. Therefore, one can choose to use it in a standards committee without getting embroiled in a long discussion over *which* computer language to use.

As a mechanism for the standards writer, then, it looks like ASN.1 is a clear winner – all positives and no negatives. The example below will show some of these advantages in operation.

2.2 The implementor

An implementor is not concerned with smoothly functioning standards committees and concise, readable standards documents. The implementor has chosen some computer language or hardware design mechanism and wants to create some program or device. The data structures which the standards writer specifies in ASN.1 have to be expressed in some form. In particular, they have to be expressed in two forms: in a computer's memory and in transit between computers.

In short, the implementor demands a definition of computer memory structures and messages – and forces ASN.1 to become a compilable language for specifying both. This destroys its main value – that it is not a computer language over which people on a standards committee might fight.

ASN.1 does not address the representation of a structure within a computer's memory, but there are associated encoding rule options (BER, DER, ...) which specify the representation of an ASN.1 structure in transit between computers[1].

A major problem, however, is that ASN.1 starts with an abstract syntax and the encoding rules must be able to handle everything ASN.1 can express.

[1]One implementor, apparently responding to the lack of an internal representation specification, held ASN.1 structures as packed byte strings within the computer's memory and parsed/re-packed those byte strings for each field access.

Since ASN.1 has no concept of size limits, the encoding rules must also be free of size limits (as must any general computer memory representation of an ASN.1 structure). Each field or substructure has a length and the length fields can express sizes as large as $(2^{1008} - 1)$ bytes. For larger values, there is an indefinite length encoding option allowing infinite length fields and structures.

This is all very cute – however, in the rush to find ways to encode abstract syntax, there was no mechanism defined to allow someone to specify an `unsigned short int` simply. So, ASN.1 itself and its encoding rules acquired a wart – a means of noting, after the definition of an INTEGER (which is supposed to be of *any* length), that the legal values are to be within a specific range.

To an implementor who, without outside interference, would have specified a data structure in some real programming language and then have specified an encoding of that structure for transferring it between machines, the ASN.1 mechanism (with its encoding rules) is a perversion.

3 An example

Let's take a concrete example.

I had an application which was doing 3-key triple-DES encryption of a data block, in CBC mode. The keying material for this encryption needed to be transmitted to another process (inside an RSA encryption) and therefore had to be packed into a byte string. I intended to have this application interoperate among machines of different endianness, so I had to be careful about machine independence. I also wanted the application to migrate to machines of different word sizes, but that was not a firm requirement.

The keying material consisted of 1 IV (of 64 bits) and up to 3 keys (of 56 bits each). However, DES keys occupy 64 bits each (with one unused bit in the least significant bit of each byte) – so from a programmer's perspective, this is a structure of four 8-byte fields.

Thinking as an implementor, I defined a C structure[2]:

```
unsigned char kiv[32] ; /* IV and 3 DES keys */
#define IV  (0)   /* offset for the IV */
#define K1  (8)   /* offset for K1 (key for 1st encr) */
#define K2 (16)   /* offset for K2 */
#define K3 (24)   /* offser for K3 */
```

Data structure packing was a non-problem, since this is already a byte string. Access to the structure was via calls such as:

```
ok = triple_des_setup( &(kiv[IV]), &(kiv[K1]),
                       &(kiv[K2]), &(kiv[K3]),
                       &des_env ) ;
```

which is all very straight-forward. The code produced is both readable and efficient. The encoding for transit is extremely efficient. There is no packing or parsing code involved.

However, this isn't modern. It doesn't use ASN.1. It isn't in proper standards form. It assumes C and that might cause fights in the committee.

So, let's do the same thing with ASN.1.

4 The ASN.1 example

First we have to decide whether to use IMPLICIT or EXPLICIT context tags. Let's choose the default – EXPLICIT[3] – not mention it again and hope it works[4].

The next step is to define a basic key and IV structure.

```
KeyAndIV ::= SEQUENCE {
```

[2]It could be argued that the IV doesn't need secrecy while the keys do, so the IV should be transmitted separately. However, it doesn't hurt security to pass the IV in secret – and if IV and keys were generated by a pseudo-random number source rather than a truly random source, then exposure of the IV might give information to the enemy about the keys themselves.

[3]... or is the default left up to the writer of the particular ASN.1 compiler you happen to use?

[4]As it turns out, we won't use context specific tags in this example.

```
    initialValue  OCTET STRING,
    key1          OCTET STRING,
    key2          OCTET STRING,
    key3          OCTET STRING
}
```

We could stop there, but we're in a standards committee and Alice points out that while an IV might be a byte string (a block), the key is really a bit string. This potential conflict is resolved by defining sub-structures for both the IV and the key, allowing them to take on data types to be agreed upon later. Bob points out that the KeyAndIV block is really the contents of a key exchange block and should be renamed. Charlie points out that the KeyExchangeBlockContents should describe what algorithm is being used for the encryption. Fortunately, ASN.1 is really easy to change. A few swipes and strokes on the whiteboard and we have:

```
KeyExchangeBlockContents ::= SEQUENCE {
    algorithmIdentifier   AlgorithmIdentifier,
    initialValue          InitialValue,
    key1                  Key,
    key2                  Key,
    key3                  Key
}

AlgorithmIdentifier ::= OBJECT IDENTIFIER

InitialValue ::= OCTET STRING

Key ::= BIT STRING
```

Now we're getting down to business in this standards meeting. We're starting to be productive.

Dave points out that we might want to make some of these keys optional. Maybe we just want to have single-DES. So, another swipe and we have:

```
KeyExchangeBlockContents ::= SEQUENCE {
    algorithmIdentifier   AlgorithmIdentifier,
```

```
  initialValue          InitialValue,
  key1                  Key,
  key2              [1] Key OPTIONAL,
  key3              [2] Key OPTIONAL
}

AlgorithmIdentifier ::= OBJECT IDENTIFIER

InitialValue ::= OCTET STRING

Key ::= BIT STRING
```

Eve suggests that someone may want to have the option of using different algorithms in each of the stages of the multiple encryption. No problem. ASN.1 is easy to change. We'll move the AlgorithmIdentifier from the KeyExchangeBlockContents to a KeyAndAlgorithm SEQUENCE.

```
KeyExchangeBlockContents ::= SEQUENCE {
  initialValue          InitialValue,
  key1                  KeyAndAlgorithm,
  key2              [1] KeyAndAlgorithm OPTIONAL,
  key3              [2] KeyAndAlgorithm OPTIONAL
}

AlgorithmIdentifier ::= OBJECT IDENTIFIER

InitialValue ::= OCTET STRING

KeyAndAlgorithm ::= SEQUENCE {
  algorithmIdentifier   AlgorithmIdentifier,
  Key                   BIT STRING
}
```

And, while we're at it, Fred notes that we don't need to restrict ourselves to a maximum of three keys. We could simplify the structure and make it more general by using a SEQUENCE OF for the keys. Gary points out that the IV might be transmitted separately from the key block, so it should be optional. Now we have:

```
KeyExchangeBlockContents ::= SEQUENCE {
  initialValue          InitialValue OPTIONAL,
  keys                  KeyList
}

AlgorithmIdentifier ::= OBJECT IDENTIFIER

InitialValue ::= OCTET STRING

KeyList ::= SEQUENCE OF  KeyAndAlgorithm

KeyAndAlgorithm ::= SEQUENCE {
  algorithmIdentifier   AlgorithmIdentifier,
  key                   BIT STRING
}
```

That looks good. Let's put that in our standard document. That should cover everyone. Now we can hand that to the implementor.

The implementor produces several files: asn_util.h, asn_util.c, prim_defs.h, kex_sizes.h, kex.h, kex_asn.h, kex.c and kex_asn.c – defining data structures, and providing the code to parse and pack ASN.1 to/from those structures, as well as code to carve, initialize and free those structures.

```
26353 asn_util.c
 6744 asn_util.h
 2056 prim_defs.h
 1733 kex.h
 2097 kex_asn.h
   66 kex_sizes.h
 4204 kex.c
11832 kex_asn.c
```

for a total code size of 55085 characters, as compared to the original 48 characters. This is an expansion in code by a factor of 1148. The byte string built for 3 DES keys is 86 bytes – only a factor of 2.7 expansion over the 32 bytes of the C-based scheme.

In this implementation, the SEQUENCE OF is implemented as an array of pointers to the sequenced structure blocks. That array is assumed to have a maximum number of elements. However, the code will probably have to be enhanced, to be true to the ASN.1, to permit acceptance of an indefinite number of key and algorithm identifier pairs.

The C structures which result from this process are:

```
typedef unsigned long   ulong ;
typedef unsigned char   uchar ;


/* Variable Length Entity */
typedef struct
{
        long    len;    /* # of bytes in value */
        uchar   *val;   /* pointer to the bytes of value */
} asn_vbl_lth ;

/* Bit string entity */
typedef struct
{
        long    len;    /* # of bytes in value */
        uchar   *val;   /* pointer to the bytes of value */
        short   nuub ;  /* # of unused bits in the LSByte */
        short   adv ;   /* TRUE if val advanced 1 byte */
} asn_bitstr ;

typedef asn_vbl_lth OBJECT_ID ;
typedef asn_vbl_lth  INTEGER ;
typedef asn_bitstr  BIT_STRING ;
typedef asn_vbl_lth  OCTET_STRING ;

typedef OBJECT_ID   AlgorithmIdentifier ;

typedef OCTET_STRING   InitialValue ;
```

```
typedef struct {
  AlgorithmIdentifier    algorithmIdentifier ;
  BIT_STRING     key ;
} KeyAndAlgorithm ;

typedef struct {
  long  n ;          /* number of elements */
  KeyAndAlgorithm *elt[ MAX_KeyList ] ;
} KeyList ;

typedef struct {
  InitialValue  *initialValue ;
  KeyList        keys ;
} KeyExchangeBlockContents ;

[...]

KeyExchangeBlockContents keyBlock ;

[...]

  /* code to test the sizes and algorithm IDs */
  /* of the IV and keys goes here */

  /* code to verify that an IV and 3 keys are present */
  /* and to give an error or branch elsewhere goes here */

  ok = triple_des_setup( keyBlock.initialValue->val,
                         keyBlock.keys.elt[0]->key.val,
                         keyBlock.keys.elt[1]->key.val,
                         keyBlock.keys.elt[2]->key.val,
                         &des_env ) ;
```

as compared to the original C structure which did our job quite adequately:

```
unsigned char kiv[32] ; /* IV and 3 DES keys */

[...]

    ok = triple_des_setup( &(kiv[IV]), &(kiv[K1]),
                           &(kiv[K2]), &(kiv[K3]),
                           &des_env ) ;
```

Of course, this use in triple_des_setup isn't the whole story. One must create the initial IV and keys. In the original scheme, I filled the 32 bytes from a random number source in one call. With the ASN.1-inspired structures, one must do 4 calls to that source and set up a great deal of auxilliary structure as well.

5 Philosophical debate

Even with that example, there are those who might argue in favor of the ASN.1 approach. After all, the argument goes, it is good to handle the limitless possibilities, especially when dealing with standards. This way, we can achieve interoperability when algorithms or key sizes change.

That argument is flawed, of course. The ASN.1 generality does not imply that an application written to use such structures will accept every algorithm and number of keys given to it[5]. Old code which was written from the ASN.1 spec is not guaranteed to be able to interoperate with new code using new algorithm options. For example, the old code might have been written before the new algorithm was defined.

Meanwhile, the ease of specifying and implementing the concrete example (3 DES keys and nothing else) allows someone to make a set of interoperable applications based on those limitations. When someone comes along with a change (eg., 3 keys for triple-IDEA), it is equally convenient to define a structure for its key exchange block, add a 1-byte algorithm identifier to the start

[5]There might be pressure to do that, but in that case what in the standards committee was "This is a generality which ASN.1 allows us to make" has been elevated into a requirement on the implementor (that he be able to generate or use arbitrary concatenations of arbitrary encryption algorithms) when there was no such requirement on the product in the first place.

of a structure (as opposed to a 9-byte uniquely allocated OBJECT IDENTIFIER) and add code to call IDEA three times. One can make several such changes with the same effort it takes to write the first ASN.1 parsing/packing code.

Of course, this debate can go on forever. I have provided only the implementor's side.

The examples generated here were run through an ASN.1 to C compiler which I wrote the last time I was forced to implement ASN.1 packing and parsing. Even with that compiler, the work of the implementor is multiplied, however. The longer names with multiple fields offer more opportunities for bugs and make the code less readable (since less of it fits on one page or screen). The additional checking (e.g., to make sure that (optional) pointer fields are present) represents additional programming effort.

6 An alternative

The flaw in ASN.1 is that it starts with abstract syntax.

It is possible to write a data structure definition language which has the same flavor as C or PASCAL (the only languages worth considering) – includes constructs which are expressible in both and omits constructs which lead to bugs (such as C's "union" construct or PASCAL's valueless "case" inside a RECORD). Specifications could be written by an implementor in such a language. When a seasoned implementor defines a data structure, it is defined with a minimum of leafiness so as to simplify the code used to access it. It will have small names, to ease typing. It will have comments to describe it. It will be simple to pack and unpack (into and out of a byte string).

What the standards writer wants is a language in which to specify a data structure (especially for transmission) uniquely and with all the generality of a real computer language.

What the implementor needs is a specification of a transmitted data structure and an automated tool for taking that specification and generating the data structures and pack/unpack code required to convert it. Such a tool is not difficult to produce. There might even be a commercial one by now. Specifications in such language can totally specify the block structure of an implementation's messages.