Network Working Group
Internet-Draft
Expires: March 2003

Keith Moore
University of Tennessee
September 2002

The Binary Low-Overhead Block Presentation Protocol

draft-moore-rescap-blob-02.txt

Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

This document is being submitted as a contribution to the IETF rescap working group. Comments regarding this internet-draft should be sent to the rescap mailing list at rescap@cs.utk.edu, or to the author at the address listed below. Requests to subscribe to the rescap mailing list should be sent to rescap-REQUEST@cs.utk.edu. Please include the document identifier draft-ietf-rescap-blob-01.txt in any comments.

Known errata of this specification, as well as sample code, will be made available at http://www.cs.utk.edu/~moore/blob/

ABSTRACT

This memo describes the Binary Low-Overhead Block (BLOB) protocol for on-the-wire presentation of data in the context of higher-level protocols. BLOB is designed to encode and decode data with low overhead on most CPUs, to be reasonably space-efficient, and for its representation to be sufficiently precise that it is suitable as a

canonical format for digital signatures.

1. Introduction

When designing applications-layer protocols there is sometimes a need to have an efficient means of encoding protocol elements or protocol data units. Existing solutions in this space may be deemed inadequate, for various reasons. For example:

- ASN.1 [2] and BER [3] are baroque both in terms of the abstract syntax and available on-the-wire representations, and complex to implement.

- ONC XDR [4] requires a stub generator and support libraries which are not easily available on all platforms, and there are subtle differences between the APIs provided by different implementations. XDR is large enough that it's not usually feasible to write your own implementation, and it's difficult to write portable code that can work with the various implementations that are deployed. Many XDR implementations have significant unnecessary processing overhead. This impairs performance of applications based on XDR and gives the protocol itself a worse reputation than it otherwise deserves.

- The design of MIME [5] was heavily influenced by the need to be able to operate over existing text-based mail systems which imposed a number of constraints. This worked out well for email, but for other applications, MIME is neither efficient in terms of storage density nor easy to parse.

- XML [6] is easier to parse than MIME, but still requires significant processing overhead. There is also a large and growing body of "culture" regarding how XML should be used, which paradoxically imposes a significant barrier to use of XML. (To be fair, MIME also has a fair amount of "culture" associated with it.) Finally, for small and regular data structures XML imposes a lot of overhead.

BLOB was designed to serve as an alternative to these presentation layers for use in representing relatively simple structures, consisting of a limited set of primitive data types, and where the structures can reasonably be contained within a single protocol data unit.

BLOB is designed with the following considerations:

- It should be easy and efficient to generate the encoded form.

- The encoded form should require minimal processing to decode, ideally being usable in-place (without allocating memory or copying) on most platforms.

- It should be easy to write programs which manipulate and exchange BLOBs, without needing significant external support in the form of libraries or stub generators.

- The structure should be easy and efficient to verify for internal consistency.

- For any structure to be represented there should be a unique (canonical) on-the-wire encoding which is always used.

- It should be reasonably space-efficient. However, this is secondary to minimizing processing overhead.

The BLOB approach is more feasible now than in years past because data representations have become more uniform across different computing platforms. Essentially all widely-used computers now support 32-bit integers, can address 32-bit integers which are not aligned on any larger boundary, use word sizes which are a multiple of 8 bits, and can directly address strings of 8-bit characters which are not aligned on any boundary larger than an octet. Such computers are termed "well-behaved" with respect to BLOB. BLOB is designed to be usable on machines which do not have these characteristics, but such machines will necessarily incur more data conversion overhead.

1.1. Notation

The word BLOB in upper case letters is used to refer to the protocol; that is, the algorithm used to define the encoding and decoding of data structures defined in this memo. The word "blob" in lower case letters refers to a data structure (sequence of octets) that has been produced by, or can be decoded by, the BLOB protocol.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document, when spelled entirely in upper case letters, are to be interpreted as described in [1].

2. BLOB Overview

A "blob" is a linear (octet-stream) encoding of some data structure, which is used as a protocol data unit within some application. The structure encoded by a blob is a collection of "components". Each of the components of a blob is either a "scalar" (meaning that the component consists of exactly one instance of that data type) or an

"array" (meaning that the component consists of a sequence of zero or more "elements" of a uniform data type).

The data types which can appear as components of a blob are: unsigned integer (32 bits in length), string (a variable-length sequence of octets with arbitrary values), or blob. Any of these types can occur as a scalar or in an array.

Since one blob can contain other blobs, complex nesting of structures is possible. However the blob encoder and decoder treat "embedded" blobs (blobs which occur as components of an outer blob) as opaque structures. For example, embedded blobs are not automatically decoded along with outer blobs, and a formatting error in an embedded blob does not create a formatting error for any blob that contains it.

"Variable-length" here means that the lengths of arrays need not be pre-determined by the protocol using BLOB. The maximum lengths of strings and arrays are constrained by the use of a 32-bit unsigned integer for the length of the blob, and the representation of offsets of data relative to the start of the blob as 32-bit unsigned integers. Lengths may be further constrained by the higher-level protocol's choice of transmission medium - for instance, if the blob must fit into a UDP datagram. The number of array elements is limited to 255 arrays of each data type, but this should be adequate for most data structures needed in network protocols.

2.1 Use of Data Types Not Supported by BLOB

The primitive types (unsigned 32-bit integer and octet string) were chosen because they represent the majority of data types used in network protocols, they are directly supported by most computer hardware, and because data types outside of this set are often specific to the higher-level protocol anyway. Having a small set of data types allows BLOB to be a compact yet self-describing encoding, which is efficient to decode and which does not require separate marshaling routines for each protocol data unit used by an application. A few additional types (in particular, single- and double-precision floating point) are being considered for future versions of BLOB. The BLOB protocol is intended to allow new primitive types to be added without changing the format of blobs that do not include these types.

When a higher-level protocol needs to use a data type that is not directly supported by BLOB, such data must be represented in terms of the available types. The higher-level protocol specification must define the representation of such data in terms of types supported by BLOB, and the conversion between the blob representation and the native format must be explicitly managed by the applications. For instance:

- A signed 32-bit integer may be transmitted as an unsigned 32-bit integer by encoding the signed integer in twos-complement format. On most modern machines no conversion will be necessary; however on machines for which the smallest integer representation is larger than 32 bits it will be necessary for the application to sign-extend the result.

- A 64-bit integer may be transmitted as two consecutive 32-bit integers (with the most significant word first), which would require that the receiving application arrange those two integers according to its native byte ordering. Alternatively a 64-bit integer may be transmitted as eight consecutive octets within a string (most significant byte first), which would require that the receiving application re-arrange those octets according to its local byte ordering.

- A multi-dimensional array may be represented as a single-dimensional array with the dimensions of the array passed as separate integer components.

- In the current version of BLOB, floating point numbers may be encoded in IEEE format and transmitted as either integers (modulo sign-extension issues) or strings (modulo alignment issues). Future versions of BLOB may support floating point numbers directly.

- A small dense set may be represented as bits within a scalar integer. A larger dense set may be encoded using individual bits of the elements of an integer array.

3. BLOB Organization

At the most basic level, the blob consists of an integer portion followed by an opaque portion. The integer portion is a sequence of unsigned 32-bit (4-octet) quantities, represented on-the-wire in network byte ("big-endian") order. The opaque portion is a sequence of 8-bit (1-octet) quantities.

The blob is separated into opaque and integer portions in order to facilitate efficient decoding on little-endian machines, or on any machine with a word size other than 32 bits. Having all of the integers within a blob co-located in a contiguous area allows an implementation to efficiently convert all of the integers to local format at the same time. Strings of octets are assumed to have the same representation on all platforms, so conversion is unlikely to be needed for the opaque portion.

The integer portion of a blob is further divided into a header, a list of array bases, and an integer pool. The header is used to store various data needed to decode the blob and check it for consistency. The array bases portion contains the offsets (positions relative to the start of the blob) of the each of the arrays in the blob (including the arrays used to store scalar components). The integer pool is used for storing integer data as well as the offsets of embedded blobs and strings.

The opaque portion is divided into a blob pool and a string pool. The blob pool is used to store embedded blobs; the string pool is used to store strings. The blob pool occurs immediately following the integer pool in order to ensure that embedded blobs are always aligned on a four-octet boundary (relative to the start of the blob).

Each embedded blob is padded with 0-3 zero octets until its length is an exact multiple of 4 octets. This ensures that all embedded blobs are aligned to 4-octet boundaries, allowing the blob decoder to assume (if the outer blob is on an aligned boundary) that each of the embedded blobs is also aligned.

Each string is padded with a single octet with a value of zero, which is not part of the string. This is for convenience when strings are used to store character data, with programming languages that use a zero-valued octet as a string terminator.

Embedded blobs are opaque to their enclosing blob and are NOT automatically parsed or decoded when the outer blob is decoded. If the receiving application wishes to examine contents of an inner blob, it must decode it separately from the enclosing blob.

A blob can have both scalar and array components. For simplicity in decoding and to eliminate some edge cases, all of the scalar integers of a blob are stored in a "scalar integer array" which immediately follows the last integer array component of the blob. Similarly, all of the scalar (embedded) blob parameters) are stored in a "scalar blob array" which immediately follows the last blob array component, and all of the scalar string parameters are stored in a "scalar string array" which follows the last string array component.

3.1 Representation of data types

In general, all components of a blob are elements of an array. A distinguished array of each type is used to store scalar components of that type. The base of any array (whether it is a numbered array component or an array used to hold scalar components) can be determined by decoding the array_counts_and_flags field of the blob header.

Since strings (and blobs) can be of varying length, an array of strings (or blobs) is represented internally by an array of integers. Each of these integers indicates the storage location (within the blob) of the contents of the string or blob. These integers are consecutive; the offset of element 2 of an array immediately follows the offset of element 1. Similarly, the array elements occupy consecutive storage - the storage occupied by string 3 of an array immediately follows that occupied by string 2. This allows the size of array N to be computed by subtracting its offset from that of the following array; this works for any numbered array. It also allows the length of element M to be computed by subtracting its offset from that of the following element; this works for elements (within bounds) of numbered arrays. The last scalar blob or string is a boundary case; these require an explicit test to correctly determine their length.

The individual components of a blob are encoded as follows:

3.1.1 integers and integer arrays

An unsigned integer is represented as a 32-bit quantity in big-endian format. All integer components appear in the integer_pool section of a blob.

An integer array is represented as zero or more contiguous 32-bit integers, that are stored within the integer_pool section of the blob. The location (or "base") of the array relative to the start of the blob is stored as a 32-bit integer offset. The base of this array is stored in the array_bases portion of the blob.

Scalar integer components a blob are encoded in a scalar integer array. The storage for the elements of this array is in the integer pool, and immediately follows the storage used by the last numbered integer array. The offset of the scalar integer array appears in the array_bases portion of the blob.

3.1.2 (embedded) blobs and blob arrays

An embedded blob component is represented as a series of octets which is an integral multiple of four octets long. The storage for embedded blobs is taken from the blob pool of the enclosing blob. An integer offset (relative to the beginning of the blob) indicates the starting location of the embedded blob. For scalar embedded blob components these offsets are encoded in a scalar blob array. This array (of blob offsets) is stored in the integer pool and immediately follows the offsets of the numbered blob arrays.

A blob array is represented as an integer base (stored in array_bases) which points to an array of integers (stored in the integer pool), each

element of which is the offset of a blob (within the blob pool).

Each embedded blob (within the blob pool) is followed by from 0-3 octets with the value zero, so that any subsequent blob will be aligned on a four-octet boundary. These padding octets are not considered part of the blob; however, the length of the inner blob (as seen from the enclosing blob) will include any padding.

3.1.3 strings and string arrays

A string is represented as a sequence of octets; these octets may have arbitrary values. The contets of strings are stored in the string_pool. An integer offset (stored in integer_pool) indicates the location of the contents of the string.

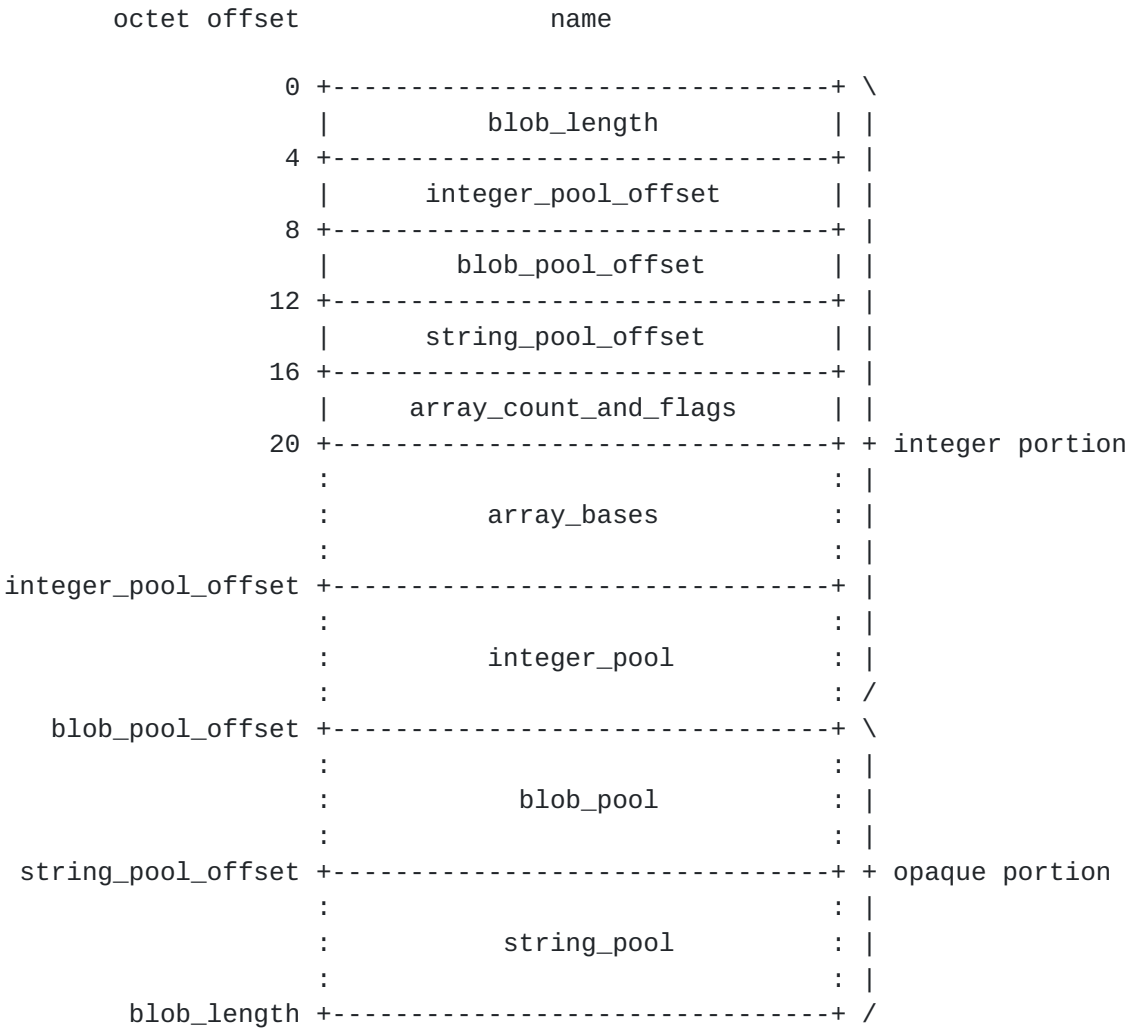
A string array is represented as an integer base (stored in array_bases) which points to an array of integers (stored in the integer pool), each element of which indicates the offset of a string (stored in string pool).

Each string is followed in the string_pool by a zero octet which is not part of the string. Thus the length of any string (other than the last scalar string component) can be calculated by subtracting its offset from the offset of the subsequent string, minus 1.

Strings can be of zero length, in which case the corresponding offset points to a zero octet which is immediately followed by the next string in the string_pool.

3.2 Structure of a blob

The structure of a blob is as follows:



For this version of the BLOB protocol, the integer portion begins at offset 0 and is blob_pool_offset octets in length. The opaque portion begins at blob_pool_offset and is (blob_length - blob_pool_offset) octets in length.

Future versions of the BLOB protocol may add additional pools for other data types, and therefore may change these formulas. BLOB decoder implementations MUST therefore decode 'array_count_and_flags' (see below) and verify that the flags portion of this field is equal to zero, before translating the remainder of the integer portion to the format used by the local machine.

The following paragraphs describe the fields within a blob:

blob_length
: The blob_length is the length of the entire blob in octets. The length includes the space occupied by blob_length. blob_length does not include any padding which is added to make an embedded blob a multiple of four octets long.

integer_pool_offset
: The integer_pool_offset is the octet offset (relative to the start of the blob) of the integer_pool field of the blob. integer_pool_offset MUST be a multiple of four, greater than or equal to 24, and less than or equal to blob_pool_offset. If the length of integer_pool is zero, integer_pool_offset will be equal to blob_pool_offset.

blob_pool_offset
: The blob_pool_offset is the offset (relative to the start of the blob) of the blob_pool field of the blob. blob_pool_offset MUST be a multiple of four, greater than or equal to integer_pool_offset, and less than or equal to string_pool_offset. If the length of the blob_pool is zero, blob_pool_offset will be equal to string_pool_offset.

string_pool_offset
: The string_pool_offset is the offset (relative to the start of the blob) of the string_pool portion of the blob. It MUST be a multiple of four, greater than or equal to blob_pool_offset, and less than or equal to blob_length. If the length of the string_pool is zero, string_pool_offset will be equal to blob_length.

array_counts_and_flags
: The array_counts_and_flags field indicates how many of each kind of array element are contained within the blob. This field is calculated as follows:

```
array_counts_and_flags = (num_int_arrays) +
                         (num_blob_arrays << 8) +
                         (num_string_arrays << 16) +
                         (flags << 24)
```

  where num_xxx_args is the number of array arguments of type xxx.

  The "flags" portion of this field is used to indicate extensions to this format. Blobs that do not use these extensions will have a flags field of zero. For this version of the BLOB protocol, the flags field MUST be zero.

array_basess

The array_bases field contains the bases (offsets relative to the start of the blob) of each of the arrays in the blob, including those arrays which contain the scalar components of the blob (using separate arrays for scalar integer, struct, and string components). Specifically the array_bases field contains, in order:

1. The base of each integer array. There are num_int_arrays (possibly zero) of these.

2. The base of the scalar integer array. This base is always present, even if there are no scalar integer components. If there are no scalar integer components of the blob, the scalar integer array base will be the same as the base of blob array 0. (If there are no blob arrays in the blob, the base of the scalar integer array will be the same as the base of the scalar blob array.)

3. The base of each blob array. There are num_blob_arrays (possibly zero) of these.

4. The base of the scalar blob array. This base is always present. If there are no embedded scalar blob components in the blob, the scalar blob array base will have the same value as the base of string array 0. (If there are no string arrays in this blob, this offset will be the same as the base of the scalar string array.)

5. The base of each string array. There are num_string_arrays (possibly zero) of these.

6. The base of the scalar string array. If there are no scalar string components of the blob, the base of the scalar string array will be equal to blob_length.

7. Any additional bases of arrays, or offsets of scalar components, which might be defined by future versions of this protocol. The presence of additional data types not supported in this version of the BLOB protocol will be indicated by a nonzero value in the flags portion of the array_counts_and_flags field.

integer_pool

The integer_pool contains 32-bit integers, assumed to be unsigned. These may be either scalar integer, elements of integer arrays, offsets of scalar blobs or strings, or bases of blob or string arrays The integers within the integer_pool MUST appear in the following order:

1. The elements of integer arrays. The integer array components appear in order, and within each array, the elements appear in order. The arrays and their elements are numbered from zero. Thus the 0th element of the 1st integer array immediately follows the last element of the 0th integer array.

2. The elements of the scalar integer array. Thus integer scalar component 0 immediately follows the last element of the last integer array; followed by integer scalar component 1, etc. (If there are no integer arrays, the offset of integer scalar 0 is integer_pool).

3. The offsets of elements of blob arrays. Each blob offset MUST be an integral multiple of four, and each blob offset MUST point into the blob_pool. The offset of the element 0 of blob array 0 MUST be equal to blob_pool_offset. Each subsequent element of a blob array MUST have an offset equal to the offset of the preceding blob plus the declared length of the preceding blob (after padding).

   NOTE: The data within an embedded blob is considered opaque to the enclosing blob; the only reason for separating blobs from strings is to ensure padding of blobs to 4-octet boundaries. Blob encoders SHOULD NOT insist that the length field of an embedded blob is consistent with the length declared for that blob, and blob decoders SHOULD NOT check the length fields of embedded blobs when decoding the enclosing blob.

4. The offsets of elements of the scalar blob array. Each blob offset MUST be a integral multiple of four, and MUST point into the blob_pool. The offset of scalar blob component 0 MUST immediately follow the last element of the last blob array. (If there are no blob arrays, the offset of scalar blob component 0 is blob_pool). Each subsequent scalar blob component MUST have an offset equal to the offset of the preceding blob plus the length of the preceding blob (after padding).

5. The offsets of elements of string arrays. These offsets MUST point into the string_pool. Element 0 of string array 0 MUST have an offset equal to string_pool_offset, and each subsequent string MUST have an offset equal to the preceding string's offset, plus the length of the preceding string, plus 1 (for the trailing zero octet).

6. The offsets of elements of the scalar string array. These offsets MUST point into the string_pool. The scalar string component 0 MUST have an offset equal to the offset of the

preceding string, plus the length of the preceding string, plus 1 (for the trailing zero octet). (If there are no string arrays, the offset of scalar string 0 is string_pool).

blob_pool

The blob_pool contains structures which are encoded in blob format. These structures may be scalar blob components of the outer blob, or elements of scalar blob arrays of the outer blob. The contents of blob_pool appear in the following order:

1. The contents of each element of each blob array. Element 0 of blob array 0 appears first, followed by element 1 of blob array 0, etc.

2. The contents of each element of the scalar blob array, used to store scalar (embedded) blob components of the outer blob.

Each blob in the blob pool MUST be padded with from zero to three octets, each with a value of zero, so that the length of each blob is an exact multiple of four octets.

string_pool

The string_pool contains unaligned strings of arbitrary octets. These strings may be used for character data or for any other data which can be represented as a string of octets. BLOB makes no assumptions regarding the format of data (character encoding scheme, etc.) that is stored in strings.

The contents of the string_pool appear in the following order:

1. The contents of each element of each string array of the blob.

2. The contents of each element of the scalar string array.

For compatibility with programming languages which terminate strings with a zero octet, a zero octet is automatically appended to each string in the string_pool. This zero octet is not part of the string. Since zero octets MAY appear within BLOB strings, the zero octet that is appended to each string MUST NOT be used as a string terminator except when the higher-level protocol has specified that they may be used in this way.

4. Use of blobs by higher-level protocols

Higher-level protocols using BLOB as an encoding mechanism need to define their protocol data units in terms of blobs. Since BLOB groups all similarly-typed data together within the blob (for ease of conversion), and since BLOB rigidly defines the order in which data must

appear, applications generally cannot refer to protocol elements within a blob by a fixed offset. Instead, the application code references protocol elements in terms of "the second scalar string component", "the third scalar integer component" or "the second element of the fourth integer array component". Macros or functions which allow these elements to be accessed from a decoded blob structure are easily constructed.

It is possible to design a simple specification language which allows the elements of a blob to be specified in the order that makes the most sense to an application, and which produces a list of macros which map from protocol data element names to routines which can access those data elements. This hides the details of BLOB's reordering from the application without significantly impairing efficiency. An example of such a language is given in Appendix B.

If higher-level protocols employ data types other than the BLOB primitive data types, they must define how the application-specific data types are represented as one or more BLOB primitive types, and implementations of the protocol will be responsible for conversion. Applications which require a canonical form (say for signing) should specify the conversion from application data types to BLOB types so that there is exactly one possible representation of each application data type within BLOB.

Since each blob is self-contained with its own header, embedded blobs add a bit of overhead. Protocol designers should avoid unnecessary nesting of structures. For instance, what is conceptually an array of structures to an application might be better represented within BLOB as several parallel arrays. However, nesting of blobs is useful when it is desired that an inner blob be opaque to the layer of a protocol that decodes the outer blob.

4.1. Encoding Issues

Most blobs will contain at least one variable-length data structure. This implies that the offsets of the components within the blob will not be known in advance, and a program that encodes a blob will usually be unable to generate the elements of a blob in-place. The encoder routine will generally need to copy the elements of a blob from their various locations into a contiguous area of memory, in the order prescribed by the BLOB specification.

4.2. Decoding Issues

On "well-behaved" machines it should be possible to use blobs in-place after converting the integer portion of the blob to the local byte order. The protocol elements within the blob can then be accessed with

macros.

It is necessary to check the blob for consistency before using it. In particular:

- The blob_length must be consistent with the length of the PDU or buffer in which the blob was received. (For instance, it must not be less than the length of data received).

- The blob_length must be at least 32 (which would be the length of an empty blob with no arguments).

- The 'flags' portion of array_counts_and_flags MUST be zero.

- The integer_pool_offset must be equal to the the number of arguments (decoded from array_counts_and_flags) multiplied by 4, plus 20.

- The blob_pool_offset must be greater than or equal to integer_pool_offset.

- The string_pool_offset must be greater than or equal to blob_pool_offset.

- The string_pool_offset must be less than or equal to blob_length.

- The base of each integer array and each blob array must be an integral multiple of 4.

- The base of the first integer array (if any) must be equal to integer_pool_offset.

- Each subsequent integer array base must be greater than or equal to the previous integer array base, and less than or equal to blob_pool_offset.

- The offset of element 0 of the first blob array (if any) must be equal to blob_pool_offset.

- Each subsequent blob offset must be greater than the previous blob offset.

- The last blob offset must be less than string_pool_offset.

- The first string component must have an offset equal to string_pool.

- The offset of each subsequent string must be greater than the offset of the first element of the previous string.

- Except for the first string, there must be a zero octet preceding each offset of each string component or string array element.

- The last octet in the string_pool must be a zero.

4.3 Encoding and decoding code

A free software sample blob encoder and decoder have been written and will be made available at the location listed in Appendix C.

5. Security Considerations

It is believed that the BLOB encoding is unique and can serve as a useful 'canonical form' for a data structure. However, if higher-level protocols encode non-native data types as BLOB primitive types, they must also define a unique representation for each quantity to be stored in that data-type.

In order to prevent possible attacks by transmission of blobs containing bogus offsets, it is essential to perform the bounds checks listed in section 4.2 while decoding blobs. While such attacks could not easily overwrite memory with data chosen by an attacker, they could cause a server to malfunction.

6. Author's Address

Keith Moore
University of Tennessee
1122 Volunteer Blvd, Suite 203
Knoxville TN 37996-3450
email: moore@cs.utk.edu

7. References

[1]. Bradner, S. "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, March 1997.

[2] "Information technology - Abstract Notation One (ASN.1): Specification of basic notation" ITU-T recommendation X.680, December 1997. Available from http://www.itu.int/ITU-T/studygroups/com17/languages/.

[3] "Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER) Canonical Encoding Rules (CER) and

Distinguished Encoding Rules (DER)" ITU-T recommendation X.690, December 1997. Available from http://www.itu.int/ITU-T/studygroups/com17/languages/.

[4] Srinivasan, R., "XDR: External Data Representation Standard", RFC 1832, August 1995.

[5] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

[6] "Extensible Markup Language (XML) 1.0 (Second Edition)", W3C Recommendation, October 2000, <http://www.w3.org/TR/2000/REC-xml-20001006>.

[7] Crocker, D. (ed.), Overell, P. "Augmented BNF for Syntax Specifications: ABNF.". RFC 2234, November 1997.

Appendix A. ASCII-Art Picture of a blob

This diagram attempts to illustrate the ordering of the various elements of a blob and the relationship of the offsets to the elements to which they point.

The following is a dump, in an assembler-like notation, of a blob which encodes:

```
2 scalar integers with values 10, 20 (decimal)
1 integer array, with elements { 1 2 3 4 }
0 scalar blobs
0 blob arrays
1 scalar string with the value "string"
2 string arrays, with elements { "a" "b" } and { "cc" "dd" "ee" }.
```

"label" denotes the name assigned to a particular offset; "xx" gives the offset in hexadecimal; "contents" gives the value of the octet or octets which appear at that offset; and "description" gives a description of the value that appears in that location.

```
                      label xx  contents description
------------------------:--:---------:-----------------------
                        :00: 00000070: blob_length
                        :04: 0000002c: integer_pool
                        :08: 0000005c: blob_pool
                        :0c: 0000005c: string_pool
                        :10: 00020002: array_count_and_flags
                        :14: 0000002c: int_array_base_0
                        :18: 0000003c: scalar_int_array_base
                        :1c: 00000044: scalar_blob_array_base
                        :20: 00000044: string_array_base_0
                        :24: 0000004c: string_array_base_1
                        :28: 00000058: scalar_string_array_base
             integer_pool:
          int_array_base_0:2c: 00000001:
                        :30: 00000002:
                        :34: 00000003:
                        :38: 00000004:
       scalar_int_array_base:3c: 0000000a: (10 decimal)
                        :40: 00000014: (20 decimal)
     scalar_blob_array_base:
        string_array_base_0:44: 0000005c: ptr_to_str[0,0]
                        :48: 0000005e: ptr_to_str[0,1]
        string_array_base_1:4c: 00000060: ptr_to_str[1,0]
                        :50: 00000063: ptr_to_str[1,1]
                        :54: 00000066: ptr_to_str[1,2]
scalar_string_array_base:58: 00000069: ptr_to_scalar_str[0]
```

```
                  blob_pool:
                string_pool:
            ptr_to_str[0,0]:5c: 61: 'a'
                            :5d: 00:
            ptr_to_str[0,1]:5e: 62: 'b'
                            :5f: 00:
            ptr_to_str[0,0]:60: 63: 'c'
                            :61: 63: 'c'
                            :62: 00:
            ptr_to_str[0,0]:63: 64: 'd'
                            :64: 64: 'd'
                            :65: 00:
            ptr_to_str[0,0]:66: 65: 'e'
                            :67: 65: 'e'
                            :68: 00:
       ptr_to_scalar_str[0]:69: 73: 's'
                            :6a: 74: 't'
                            :6b: 72: 'r'
                            :6c: 69: 'i'
                            :6d: 6e: 'n'
                            :6e: 67: 'g'
                            :6f: 00:
                blob_length:70:
```

Appendix B. Example Abstract Syntax

This syntax used to describe BLOB structures is described below using the ABNF syntax from [7]:

```
file = *(block / comment-line)

block = "BEGIN" 1*space id [ 1*space comment ] CRLF
        *element
        END [ comment ] CRLF

element = "int" 1*space identifier [ comment ] CRLF /
          "string" 1*space identifier [ comment ] CRLF /
          "int<>" 1*space identifier [ comment ] CRLF /
          "string<>" 1*space identifier [ comment ] CRLF /
          "struct" 1*space identifier [ comment ] CRLF
          "struct<>" 1*space identifier [ comment ] CRLF

comment = *space "#" *char

comment-line = comment CRLF

id = letter *(letter / digit / "_")

letter = "A".."Z"        # includes lower case also

digit = "0".."9"

space = %20 / %09

char = %01..%09 / %0B / %0C / %0E..%FF

CRLF = 0*1%0D 0*1%0A
```

Here is a simple awk program to interpret this syntax and produce a list of C #define macros. The macros are of the form

```
#define structname_element_type number
```

where 'structname' is the name of the structure, 'element' is the name of the element, and 'type' is a suffix indicating the type of the element (i = int, b = blob, s = string, ia = integer array, ba = blob array, sa = string array) for ease in visual type checking.

This program is quite simplistic and performs no error checking.

```
#!/bin/sh
# the sed line deletes comments
sed -e 's/[ ]*#.*//' | awk '
$1 == "BEGIN" {
        current_id = $2;
        nint = nblob = nstr = ninta = nbloba = nstra = 0;
}
$1 == "int" {
        inames[nint] = $2;
        nint++;
        next;
}
$1 == "string" {
        snames[nstr] = $2;
        nstr++;
        next;
}
$1 == "struct" {
        bnames[nblob] = $2;
        nblob++;
        next;
}
$1 == "int<>" {
        ianames[ninta] = $2;
        ninta++;
        next;
}
$1 == "string<>" {
        sanames[nstra] = $2;
        nstra++;
        next;
}
$1 == "struct<>" {
        banames[nbloba] = $2;
        nbloba++;
        next;
}
$1 == "END" {
        for (i = 0; i < nint; ++i)
                printf ("#define %s_%s_i %d\n", current_id, inames[i], i);
        for (i = 0; i < nblob; ++i)
                printf ("#define %s_%s_b %d\n", current_id, bnames[i], i);
        for (i = 0; i < nstr; ++i)
                printf ("#define %s_%s_s %d\n", current_id, snames[i], i);
        for (i = 0; i < ninta; ++i)
                printf ("#define %s_%s_ia %d\n", current_id, ianames[i], i);
        for (i = 0; i < nbloba; ++i)
                printf ("#define %s_%s_ba %d\n", current_id, banames[i], i);
```

```
        for (i = 0; i < nstra; ++i)
                printf ("#define %s_%s_sa %d\n", current_id, sanames[i], i);
        next;
}'
```

Appendix C. Example Encoding and Decoding Code

Check http://www.cs.utk.edu/~moore/blob for the latest version.