3

NETWORK PROTOCOL STRUCTURES

The old adage "There is nothing new under the sun" holds true when it comes to the way protocols are structured. Binary and text protocols follow common patterns and structures and, once understood, can easily be applied to any new protocol. This chapter details some of these structures and formalizes the way I'll represent them throughout the rest of this book.

In this chapter, I discuss many of the common types of protocol structures. Each is described in detail along with how it is represented in binary- or text-based protocols. By the end of the chapter, you should be able to easily identify these common types in any unknown protocol you analyze.

Once you understand how protocols are structured, you'll also see patterns of exploitable behavior—ways of attacking the network protocol itself. Chapters 8 and 10 will provide more detail on finding network protocol issues, but for now we'll just concern ourselves with structure.

Binary Protocol Structures

Binary protocols work at the binary level; the smallest unit of data is a single binary digit. Dealing with single bits is difficult, so we'll use 8-bit units called *octets*, commonly called *bytes*. The octet is the de facto unit of network protocols. Although octets can be broken down into individual bits (for example, to represent a set of flags), we'll treat all network data in 8-bit units, as shown in Figure 3-1.

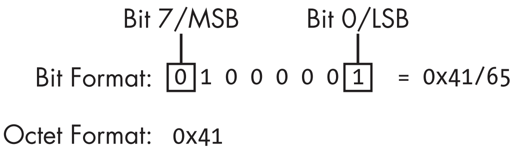


Figure 3-1: Binary data description formats

When showing individual bits, I'll use the *bit format*, which shows bit 7, the *most significant bit (MSB)*, on the left. Bit 0, or the *least significant bit (LSB)*, is on the right. (Some architectures, such as PowerPC, define the bit numbering in the opposite direction.)

Numeric Data

Data values representing numbers are usually at the core of a binary protocol. These values can be integers or decimal values. Numbers can be used to represent the length of data, to identify tag values, or simply to represent a number.

In binary, numeric values can be represented in a few different ways, and a protocol's method of choice depends on the value it's representing. The following sections describe some of the more common formats.

Unsigned Integers

Unsigned integers are the most obvious representation of a binary number. Each bit has a specific value based on its position, and these values are added together to represent the integer. Table 3-1 shows the decimal and hexadecimal values for an 8-bit integer.

Table 3-1: Decimal Bit Values

Bit	Decimal value	Hex value
0	1	0x01
1	2	0x02
2	4	0x04
3	8	0x08
4	16	0x10
5	32	0x20
6	64	0x40
7	128	0x80

Signed Integers

Not all integer values are positive. In some scenarios, negative integers are required—for example, to represent the difference between two integers, you need to take into account that the difference could be negative—and only signed integers can hold negative values. While encoding an unsigned integer seems obvious, the CPU can only work with the same set of bits. Therefore, the CPU requires a way of interpreting the unsigned integer value as signed; the most common signed interpretation is two's complement. The term *two's complement* refers to the way in which the signed integer is represented within a native integer value in the CPU.

Conversion between unsigned and signed values in two's complement is done by taking the bitwise NOT of the integer and adding 1. For example, Figure 3-2 shows the 8-bit integer 123 converted to its two's complement representation.

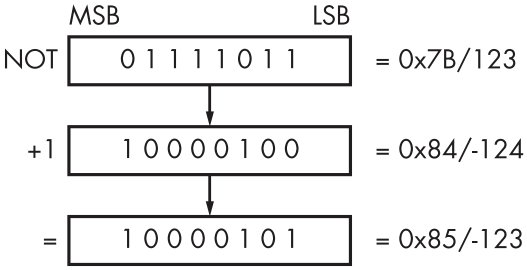


Figure 3-2: Two's complement representation of 123

The two's complement representation has one dangerous security consequence. For example, an 8-bit signed integer has the range –128 to 127, so the magnitude of the minimum is larger than the maximum. If the minimum value is negated, the result is itself; in other words, –(–128) is –128. This can cause calculations to be incorrect in parsed formats, leading to security vulnerabilities. We'll go into more detail in Chapter 10.

Variable-Length Integers

Efficient transfer of network data has historically been very important. Even though today's high-speed networks might make efficiency concerns unnecessary, there are still advantages to reducing a protocol's bandwidth. It can be beneficial to use variable-length integers when the most common integer values being represented are within a very limited range. This is in contrast to large integers that vary in size to extend their maximum range.

For example, consider length fields: when sending blocks of data between 0 and 127 bytes in size, you could use a 7-bit variable integer representation. Figure 3-3 shows a few different encodings for 32-bit words. At most, five octets are required to represent the entire range. But if your protocol tends to assign values between 0 and 127, it will only use one octet, which saves a considerable amount of space.

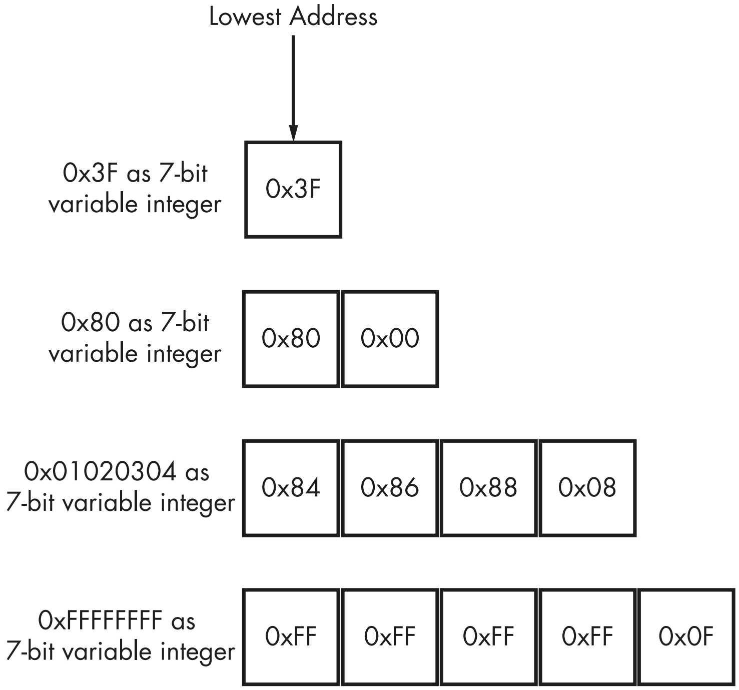

Lowest Address
0x3F as 7-bit variable integer
0x3F
0x80 as 7-bit variable integer
0x80
0x00
0x01020304 as 7-bit variable integer
0x84
0x86
0x88
0x08
0xFFFFFFFF as 7-bit variable integer
0xFF
0xFF
0xFF
0xFF
0x0F

Figure 3-3: Example 7-bit integer encoding

That said, if you parse more than five octets (or even 32 bits), the resulting integer from the parsing operation will depend on the parsing program. Some programs (including those developed in C) will simply drop any bits beyond a given range, whereas other development environments will generate an overflow error. If not handled correctly, this integer overflow might lead to vulnerabilities, such as buffer overflows, which could cause a smaller than expected memory buffer to be allocated, in turn resulting in memory corruption.

Floating-Point Data

Sometimes, integers aren't enough to represent the range of decimal values needed for a protocol. For example, a protocol for a multiplayer computer game might require sending the coordinates of players or objects in the game's virtual world. If this world is large, it would be easy to run up against the limited range of a 32- or even 64-bit fixed-point value.

The format of floating-point integers used most often is the *IEEE format* specified in IEEE Standard for Floating-Point Arithmetic (IEEE 754). Although the standard specifies a number of different binary and even decimal formats for floating-point values, you're likely to encounter only two: a single-precision binary representation, which is a 32-bit value; and a double-precision, 64-bit value. Each format specifies the position and bit size of the significand and exponent. A sign bit is also specified, indicating whether the value is positive or negative. Figure 3-4 shows the general layout of an IEEE floating-point value, and Table 3-2 lists the common exponent and significand sizes.

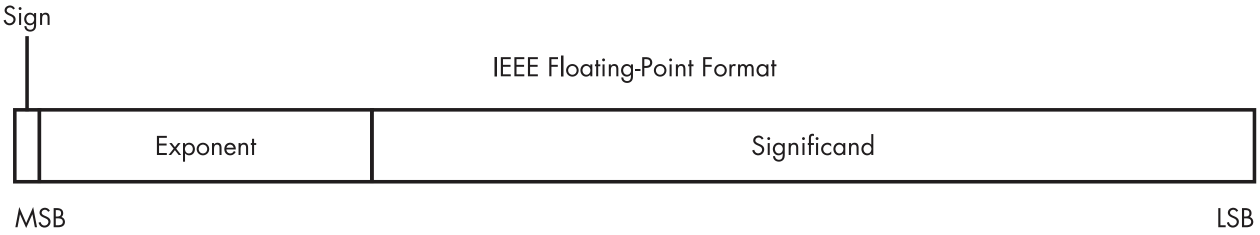


Figure 3-4: Floating-point representation

Table 3-2: Common Float Point Sizes and Ranges

Bit size	Exponent bits	Significand bits	Value range
32	8	23	+/– 3.402823×10^{38}
64	11	52	+/– $1.79769313486232 \times 10^{308}$

Booleans

Because Booleans are very important to computers, it's no surprise to see them reflected in a protocol. Each protocol determines how to represent whether a Boolean value is true or false, but there are some common conventions.

The basic way to represent a Boolean is with a single-bit value. A 0 bit means false and a 1 means true. This is certainly space efficient but not necessarily the simplest way to interface with an underlying application. It's more common to use a single byte for a Boolean value because it's far easier to manipulate. It's also common to use zero to represent false and non-zero to represent true.

Bit Flags

Bit flags are one way to represent specific Boolean states in a protocol. For example, in TCP a set of bit flags is used to determine the current state of a connection. When making a connection, the client sends a packet with the synchronize flag (SYN) set to indicate that the connections should synchronize their timers. The server can then respond with an acknowledgment (ACK) flag to indicate it has received the client request as well as the SYN flag to establish the synchronization with the client. If this handshake used single enumerated values, this dual state would be impossible without a distinct SYN/ACK state.

Strings

Along with numeric data, strings are the value type you'll most commonly encounter, whether they're being used for passing authentication credentials or resource paths. When inspecting a protocol designed to send only English characters, the text will probably be encoded using ASCII. The original ASCII standard defined a 7-bit character set from 0 to 0x7F, which includes most of the characters needed to represent the English language (shown in Figure 3-5).

Control Character | Printable Character

Lower 4 bits

Upper 4 bits	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Figure 3-5: 7-bit ASCII table

The ASCII standard was originally developed for text terminals (physical devices with a moving printing head). Control characters were used to send messages to the terminal to move the printing head or to synchronize serial communications between the computer and the terminal. The ASCII character set contains two types of characters: *control* and *printable*. Most of the control characters are relics of those devices and are virtually unused. But some still provide information on modern computers, such as CR and LF, which are used to end lines of text.

The printable characters are the ones you can see. This set of characters consists of many familiar symbols and alphanumeric characters; however, they won't be of much use if you want to represent international characters, of which there are thousands. To address this problem, other ways of encoding (such as UTF-8) are becoming prevalent, as you'll learn later in this chapter.

Character Encoding

Life would be easy in the text-based protocol world if everyone used ASCII character encoding to represent textual data. Unfortunately, it's unachievable to represent even a fraction of the possible characters in all the world's languages in a 7-bit number.

Three strategies are commonly employed to counter this limitation: code pages, multibyte character sets, and Unicode. A protocol will either require that you use one of these three ways to represent text, or it will offer an option that an application can select.

Code Pages

The simplest way to extend the ASCII character set is by recognizing that if all your data is stored in octets, 128 unused values (from 128 to 255) can be repurposed for storing extra characters. Although 256 values are not enough to store all the characters in every available language, you have many different ways to use the unused range. Which characters are mapped to which values is typically codified in specifications called *code pages* or *character encodings*.

Multibyte Character Sets

In languages such as Chinese, Japanese, and Korean (collectively referred to as CJK), you simply can't come close to representing the entire written language with 256 characters, even if you use all available space. The solution is to use multibyte character sets combined with ASCII to encode these languages. Common encodings are Shift-JIS for Japanese and GB2312 for simplified Chinese.

Multibyte character sets allow you to use two or more octets in sequence to encode a desired character, although you'll rarely see them in use. In fact, if you're not working with CJK, you probably won't see them at all. (For the sake of brevity, I won't discuss multibyte character sets any further; plenty of online resources will aid you in decoding them if required.)

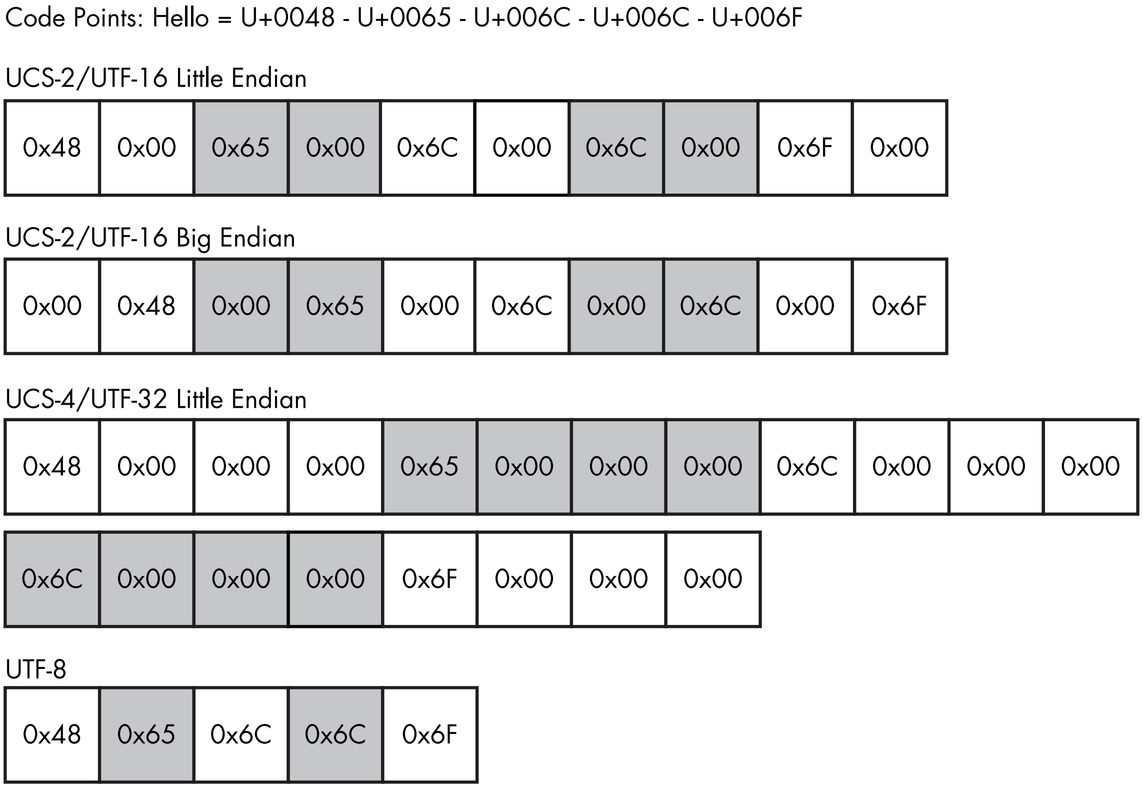
Unicode

The Unicode standard, first standardized in 1991, aims to represent all languages within a unified character set. You might think of Unicode as another multibyte character set. But rather than focusing on a specific language, such as Shift-JIS does with Japanese, it tries to encode all written languages, including some archaic and constructed ones, into a single universal character set.

Character Mapping and Character Encoding

Unicode defines two related concepts: *character mapping* and *character encoding*. *Character mappings* include mappings between a numeric value and a character, as well as many other rules and regulations on how characters are used or combined. *Character encodings* define the way these numeric values are encoded in the underlying file or network protocol. For analysis purposes, it's far more important to know how these numeric values are encoded.

Each character in Unicode is assigned a *code point* that represents a unique character. Code points are commonly written in the format *U+ABCD*, where ABCD is the code point's hexadecimal value. For the sake of compatibility, the first 128 code points match what is specified in ASCII, and the second 128 code points are taken from ISO/IEC 8859-1. The resulting value is encoded using a specific scheme, sometimes referred to as *Universal Character Set (UCS)* or *Unicode Transformation Format (UTF)* encodings. (Subtle differences exist between UCS and UTF formats, but for the sake of identification and manipulation, these differences are unimportant.) Figure 3-6 shows a simple example of some different Unicode formats.



Code Points: Hello = U+0048 - U+0065 - U+006C - U+006C - U+006F

UCS-2/UTF-16 Little Endian

0x48	0x00	0x65	0x00	0x6C	0x00	0x6C	0x00	0x6F	0x00

UCS-2/UTF-16 Big Endian

0x00	0x48	0x00	0x65	0x00	0x6C	0x00	0x6C	0x00	0x6F

UCS-4/UTF-32 Little Endian

0x48	0x00	0x00	0x00	0x65	0x00	0x00	0x00	0x6C	0x00	0x00	0x00
0x6C	0x00	0x00	0x00	0x6F	0x00	0x00	0x00				

UTF-8

0x48	0x65	0x6C	0x6C	0x6F

Figure 3-6: The string "`Hello`" in different Unicode encodings

Three common Unicode encodings in use are UTF-16, UTF-32, and UTF-8.

UCS-2/UTF-16

UCS-2/UTF-16 is the native format on modern Microsoft Windows platforms, as well as the Java and .NET virtual machines when they are running code. It encodes code points in sequences of 16-bit integers and has little and big endian variants.

UCS-4/UTF-32

UCS-4/UTF-32 is a common format used in Unix applications because it's the default wide-character format in many C/C++ compilers. It encodes code points in sequences of 32-bit integers and has different endian variants.

UTF-8

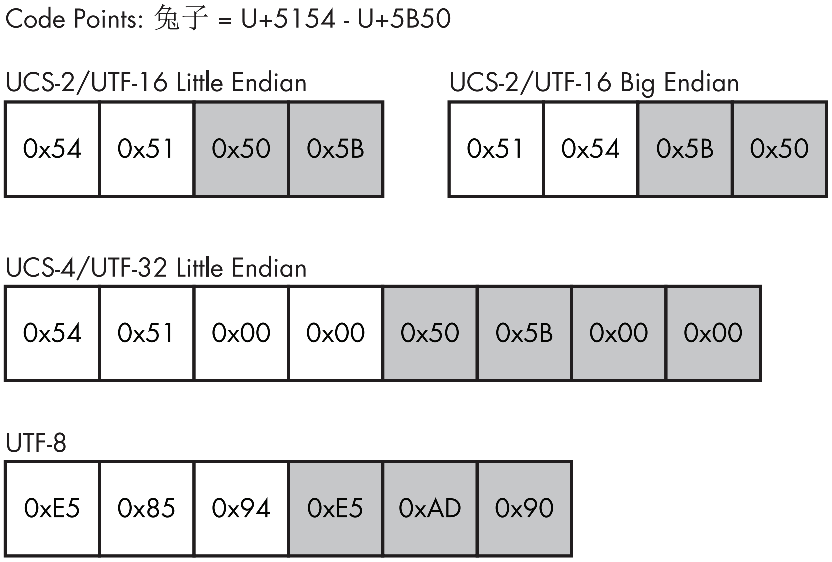
UTF-8 is probably the most common format on Unix. It is also the default input and output format for varying platforms and technologies, such as XML. Rather than having a fixed integer size for code points, it encodes them using a simple variable length value. Table 3-3 shows how code points are encoded in UTF-8.

Table 3-3: Encoding Rules for Unicode Code Points in UTF-8

Bits of code point	First code point (U+)	Last code point (U+)	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
0-7	0000	007F	0xxxxxxx					
8-11	0080	07FF	110xxxxx	10xxxxxx				
12-16	0800	FFFF	1110xxxx	10xxxxxx	10xxxxxx			
17-21	10000	1FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
22-26	200000	3FFFFFF	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
26-31	4000000	7FFFFFFF	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

UTF-8 has many advantages. For one, its encoding definition ensures that the ASCII character set, code points U+0000 through U+007F, are encoded using single bytes. This scheme makes this format not only ASCII compatible but also space efficient. In addition, UTF-8 is compatible with C/C++ programs that rely on NUL-terminated strings.

For all of its benefits, UTF-8 does come at a cost, because languages like Chinese and Japanese consume more space than they do in UTF-16. Figure 3-7 shows such a disadvantageous encoding of Chinese characters. But notice that the UTF-8 in this example is still more space efficient than the UTF-32 for the same characters.



Code Points: 兔子 = U+5154 - U+5B50

UCS-2/UTF-16 Little Endian

0x54	0x51	0x50	0x5B

UCS-2/UTF-16 Big Endian

0x51	0x54	0x5B	0x50

UCS-4/UTF-32 Little Endian

0x54	0x51	0x00	0x00	0x50	0x5B	0x00	0x00

UTF-8

0xE5	0x85	0x94	0xE5	0xAD	0x90

Figure 3-7: The string "兔子" in different Unicode encodings

NOTE *Incorrect or naive character encoding can be a source of subtle security issues, ranging from bypassing filtering mechanisms (say in a requested resource path) to causing buffer overflows. We'll investigate some of the vulnerabilities associated with character encoding in Chapter 10.*

Variable Binary Length Data

If the protocol developer knows in advance exactly what data must be transmitted, they can ensure that all values within the protocol are of a fixed length. In reality this is quite rare, although even simple authentication credentials would benefit from the ability to specify variable username and password string lengths. Protocols use several strategies to produce variable-length data values: I discuss the most common—terminated data, length-prefixed data, implicit-length data, and padded data—in the following sections.

Terminated Data

You saw an example of variable-length data when variable-length integers were discussed earlier in this chapter. The variable-length integer value was terminated when the octet's MSB was 0. We can extend the concept of terminating values further to elements like strings or data arrays.

A terminated data value has a terminal symbol defined that tells the data parser that the end of the data value has been reached. The terminal symbol is used because it's unlikely to be present in typical data, ensuring that the value isn't terminated prematurely. With string data, the terminating value can be a NUL value (represented by 0) or one of the other control characters in the ASCII set.

If the terminal symbol chosen occurs during normal data transfer, you need to use a mechanism to escape these symbols. With strings, it's common to see the terminating character either prefixed with a backslash (\) or repeated twice to prevent it from being identified as the terminal symbol. This approach is especially useful when a protocol doesn't know ahead of time how long a value is—for example, if it's generated dynamically. Figure 3-8 shows an example of a string terminated by a NUL value.

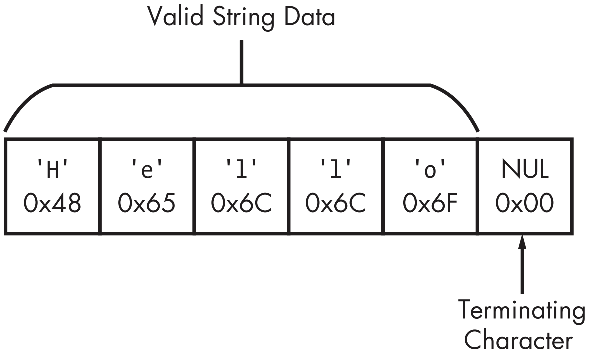


Figure 3-8: "Hello" as a NUL-terminated string

Bounded data is often terminated by a symbol that matches the first character in the variable-length sequence. For example, when using string data, you might find a *quoted string* sandwiched between quotation marks. The initial double quote tells the parser to look for the matching character to end the data. Figure 3-9 shows a string bounded by a pair of double quotes.

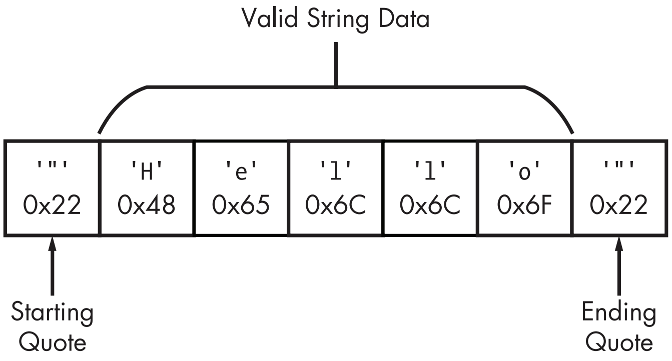


Figure 3-9: "`Hello`" as a double-quoted bounded string

Length-Prefixed Data

If a data value is known in advance, it's possible to insert its length into the protocol directly. The protocol's parser can read this value and then read the appropriate number of units (say characters or octets) to extract the original value. This is a very common way to specify variable-length data.

The actual size of the *length* prefix is usually not that important, although it should be reasonably representative of the types of data being transmitted. Most protocols won't need to specify the full range of a 32-bit integer; however, you'll often see that size used as a length field, if only because it fits well with most processor architectures and platforms. For example, Figure 3-10 shows a string with an 8-bit length prefix.

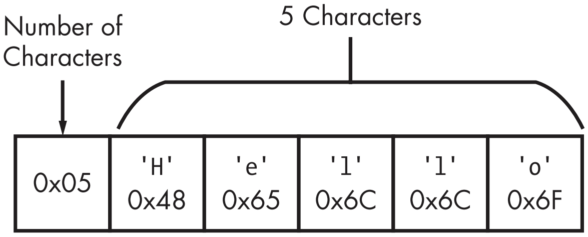


Figure 3-10: "`Hello`" as a length-prefixed string

Implicit-Length Data

Sometimes the length of the data value is implicit in the values around it. For example, think of a protocol that is sending data back to a client using a connection-oriented protocol such as TCP. Rather than specifying the size of the data up front, the server could close the TCP connection, thus implicitly signifying the end of the data. This is how data is returned in an HTTP version 1.0 response.

Another example would be a higher-level protocol or structure that has already specified the length of a set of values. The parser might extract that higher-level structure first and then read the values contained within it. The protocol could use the fact that this structure has a finite length associated with it to implicitly calculate the length of a value in a similar fashion to close the connection (without closing it, of course). For example, Figure 3-11 shows a trivial example where a 7-bit variable integer and string are contained within a single block. (Of course, in practice, this can be considerably more complex.)

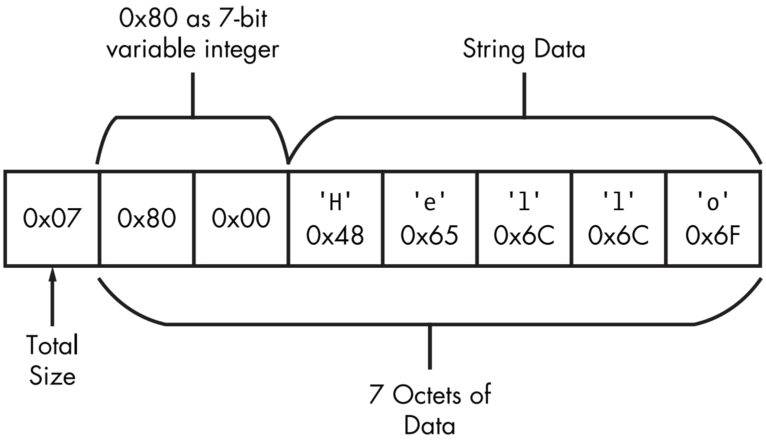


Figure 3-11: "`Hello`" as an implicit-length string

Padded Data

Padded data is used when there is a maximum upper bound on the length of a value, such as a 32-octet limit. For the sake of simplicity, rather than prefixing the value with a length or having an explicit terminating value, the protocol could instead send the entire fixed-length string but terminate the value by padding the unused data with a known value. Figure 3-12 shows an example.

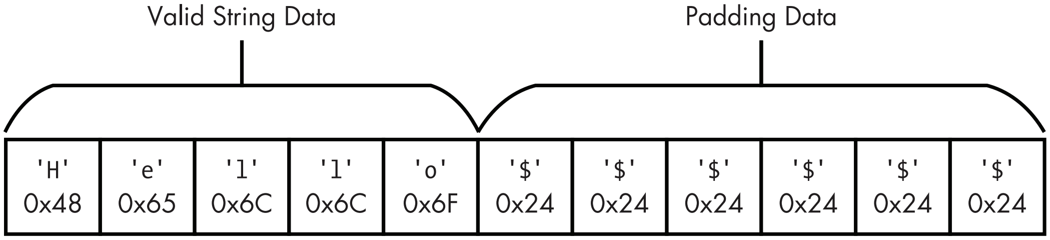


Figure 3-12: "`Hello`" as a '`$`' padded string

Dates and Times

It can be very important for a protocol to get the correct date and time. Both can be used as metadata, such as file modification timestamps in a network file protocol, as well as to determine the expiration of authentication credentials. Failure to correctly implement the timestamp might cause serious security issues. The method of date and time representation depends on usage requirements, the platform the applications are running on, and the protocol's space requirements. I discuss two common representations, POSIX/Unix Time and Windows FILETIME, in the following sections.

POSIX/Unix Time

Currently, POSIX/Unix time is stored as a 32-bit signed integer value representing the number of seconds that have elapsed since the Unix epoch, which is usually specified as 00:00:00 (UTC), 1 January 1970. Although this isn't a high-definition timer, it's sufficient for most scenarios. As a 32-bit integer, this value is limited to 03:14:07 (UTC) 19 January 2038, at which point the representation will overflow. Some modern operating systems now use a 64-bit representation to address this problem.

Windows FILETIME

The Windows FILETIME is the date and time format used by Microsoft Windows for its filesystem timestamps. As the only format on Windows with simple binary representation, it also appears in a few different protocols.

The FILETIME format is a 64-bit unsigned integer. One unit of the integer represents a 100 ns interval. The epoch of the format is 00:00:00 (UTC), 1 January 1601. This gives the FILETIME format a larger range than the POSIX/Unix time format.

Binary Endian

The endianness of data is a very important part of interpreting binary protocols correctly. It comes into play whenever a multi-octet value, such as a 32-bit word, is transferred. The endian is an artifact of how computers store data in memory.

Because octets are transmitted sequentially on the network, it's possible to send the most significant octet of a value as the first part of the transmission, as well as the reverse—send the least significant octet first. The order in which octets are sent determines the endian of the data. Failure to correctly handle the endian format can lead to subtle bugs in the parsing of protocols.

Modern platforms use two main endian formats: big and little. *Big endian* stores the most significant byte at the lowest address, whereas *little endian* stores the least significant byte in that location. Figure 3-13 shows how the 32-bit integer 0x01020304 is stored in both forms.

The endian of a value is commonly referred to as either *network order* or *host order*. Because the Internet RFCs invariably use big endian as the preferred type for all network protocols they specify (unless there are legacy reasons for doing otherwise), big endian is referred as network order. But your computer could be either big or little endian. Processor architectures such as x86 use little endian; others such as SPARC use big endian.

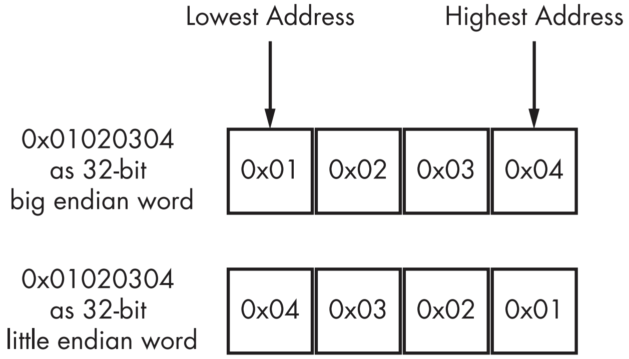


Figure 3-13: Big and little endian word representation

NOTE *Some processor architectures, including SPARC, ARM, and MIPS, may have onboard logic that specifies the endianness at runtime, usually by toggling a processor control flag. When developing network software, make no assumptions about the endianness of the platform you might be running on. The networking API used to build an application will typically contain convenience functions for converting to and from these orders. Other platforms, such as PDP-11, use a middle endian format where 16-bit words are swapped; however, you're unlikely to ever encounter one in everyday life, so don't dwell on it.*

Tag, Length, Value Pattern

It's easy to imagine how one might send unimportant data using simple protocols, but sending more complex and important data takes some explaining. For example, a protocol that can send different types of structures must have a way to represent the bounds of a structure and its type.

One way to represent data is with a *Tag, Length, Value (TLV) pattern*. The Tag value represents the type of data being sent by the protocol, which is commonly a numeric value (usually an enumerated list of possible values). But the Tag can be anything that provides the data structures with a unique pattern. The Length and Value are variable-length values. The order in which the values appear isn't important; in fact, the Tag might be part of the Value. Figure 3-14 show a couple of ways these values could be arranged.

The Tag value sent can be used to determine how to further process the data. For example, given two types of Tags, one that indicates the authentication credentials to the application and another that represents a message being transmitted to the parser, we must be able to distinguish between the two types of data. One big advantage to this pattern is that it allows us to extend a protocol without breaking applications that have not been updated to support the updated protocol. Because each structure is sent with an associated Tag and Length, a protocol parser could ignore the structures that it doesn't understand.

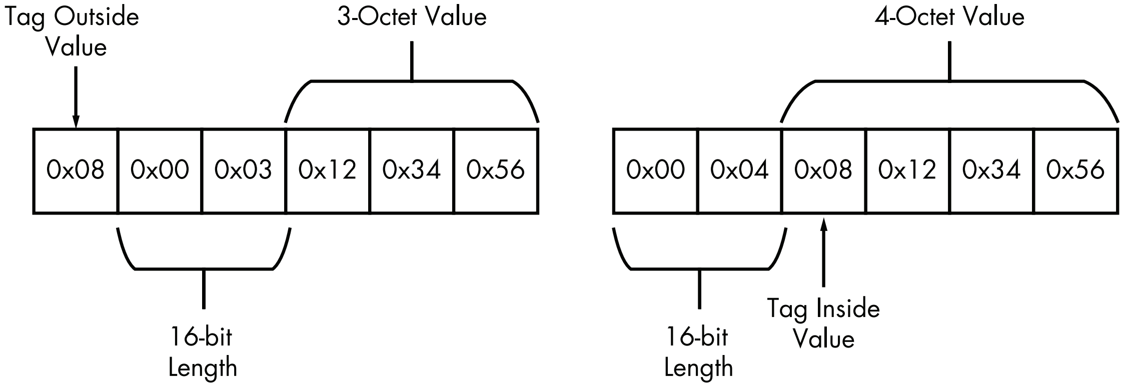


Figure 3-14: Possible TLV arrangements

Multiplexing and Fragmentation

Often in computer communication, multiple tasks must happen at once. For example, consider the Microsoft *Remote Desktop Protocol (RDP)*: a user could be moving the mouse cursor, typing on the keyboard, and transferring files to a remote computer while changes in the display and audio are being transmitted back to the user (see Figure 3-15).

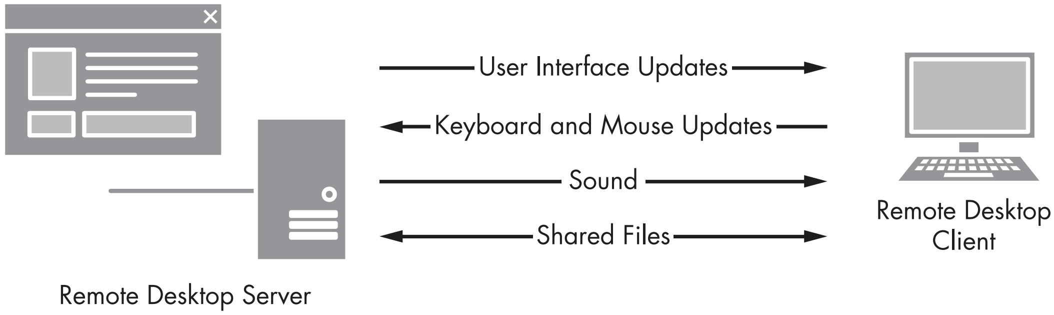


Figure 3-15: Data needs for Remote Desktop Protocol

This complex data transfer would not result in a very rich experience if display updates had to wait for a 10-minute audio file to finish before updating the display. Of course, a workaround would be opening multiple connections to the remote computer, but those would use more resources. Instead, many protocols use *multiplexing*, which allows multiple connections to share the same underlying network connection.

Multiplexing (shown in Figure 3-16) defines an internal *channel* mechanism that allows a single connection to host multiple types of traffic by fragmenting large transmissions into smaller chunks. Multiplexing then combines these chunks into a single connection. When analyzing a protocol, you may need to demultiplex these channels to get the original data back out.

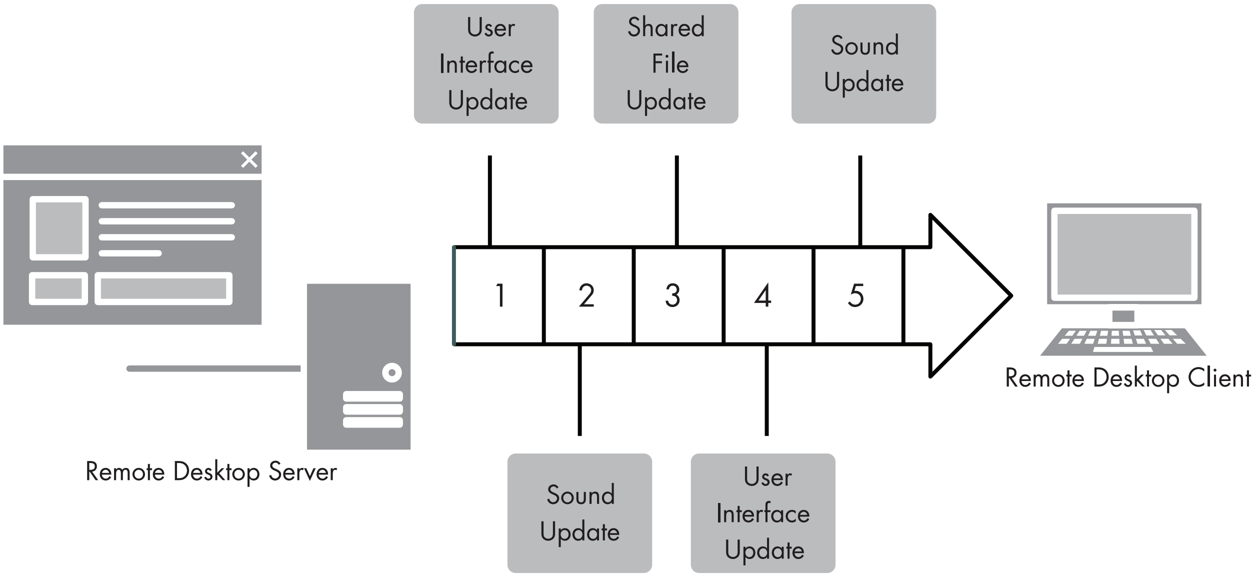


Figure 3-16: Multiplexed RDP data

Unfortunately, some network protocols restrict the type of data that can be transmitted and how large each packet of data can be—a problem commonly encountered when layering protocols. For example, Ethernet defines the maximum size of traffic frames as 1500 octets, and running IP on top of that causes problems because the maximum size of IP packets can be 65536 bytes. *Fragmentation* is designed to solve this problem: it uses a mechanism that allows the network stack to convert large packets into smaller fragments when the application or OS knows that the entire packet cannot be handled by the next layer.

Network Address Information

The representation of network address information in a protocol usually follows a fairly standard format. Because we're almost certainly dealing with TCP or UDP protocols, the most common binary representation is the IP address as either a 4- or 16-octet value (for IPv4 or IPv6) along with a 2-octet port. By convention, these values are typically stored as big endian integer values.

You might also see hostnames sent instead of raw addresses. Because hostnames are just strings, they follow the patterns used for sending variable-length strings, which was discussed earlier in "Variable Binary Length Data" on page 36. Figure 3-17 shows how some of these formats might appear.

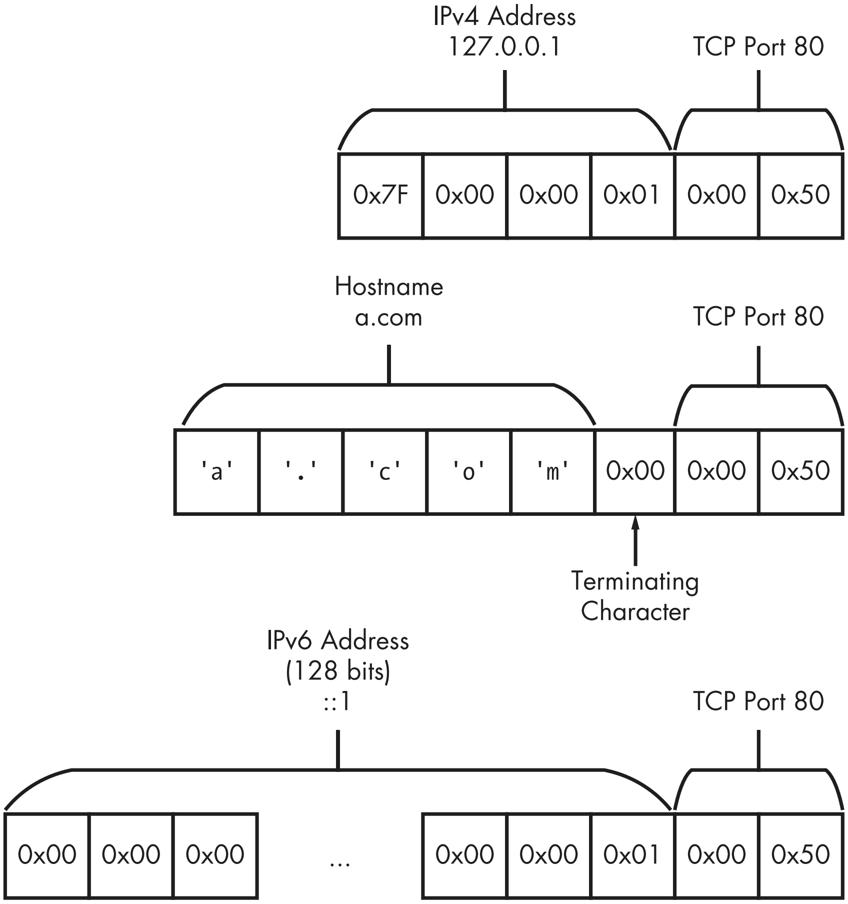


Figure 3-17: Network information in binary

Structured Binary Formats

Although custom network protocols have a habit of reinventing the wheel, sometimes it makes more sense to repurpose existing designs when describing a new protocol. For example, one common format encountered in binary protocols is *Abstract Syntax Notation 1 (ASN.1)*. ASN.1 is the basis for protocols such as the Simple Network Management Protocol (SNMP), and it is the encoding mechanism for all manner of cryptographic values, such as X.509 certificates.

ASN.1 is standardized by the ISO, IEC, and ITU in the X.680 series. It defines an abstract syntax to represent structured data. Data is represented in the protocol depending on the encoding rules, and numerous encodings exist. But you're most likely to encounter the *Distinguished Encoding Rules (DER)*, which is designed to represent ASN.1 structures in a way that cannot be misinterpreted—a useful property for cryptographic protocols. The DER representation is a good example of a TLV protocol.

Rather than going into great detail about ASN.1 (which would take up a fair amount of this book), I give you Listing 3-1, which shows the ASN.1 for X.509 certificates.

```
Certificate  ::=  SEQUENCE  {
     version         [0]  EXPLICIT Version DEFAULT v1,
     serialNumber         CertificateSerialNumber,
     signature            AlgorithmIdentifier,
     issuer               Name,
     validity             Validity,
     subject              Name,
     subjectPublicKeyInfo SubjectPublicKeyInfo,
     issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
     subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
     extensions      [3]  EXPLICIT Extensions OPTIONAL
}
```

Listing 3-1: X.509 ASN.1 representation

This abstract definition of an X.509 certificate can be represented in any of ASN.1's encoding formats. Listing 3-2 shows a snippet of the DER encoded form dumped as text using the OpenSSL utility.

```
$ openssl asn1parse -in example.cer
    0:d=0  hl=4 l= 539 cons: SEQUENCE
    4:d=1  hl=4 l= 388 cons: SEQUENCE
    8:d=2  hl=2 l=   3 cons: cont [ 0 ]
   10:d=3  hl=2 l=   1 prim: INTEGER         :02
   13:d=2  hl=2 l=  16 prim: INTEGER         :19BB8E9E2F7D60BE48BFE6840B50F7C3
   31:d=2  hl=2 l=  13 cons: SEQUENCE
   33:d=3  hl=2 l=   9 prim: OBJECT          :sha1WithRSAEncryption
   44:d=3  hl=2 l=   0 prim: NULL
   46:d=2  hl=2 l=  17 cons: SEQUENCE
   48:d=3  hl=2 l=  15 cons: SET
   50:d=4  hl=2 l=  13 cons: SEQUENCE
   52:d=5  hl=2 l=   3 prim: OBJECT           :commonName
   57:d=5  hl=2 l=   6 prim: PRINTABLESTRING  :democa
```

Listing 3-2: Small sample of X.509 certificate

Text Protocol Structures

Text protocols are a good choice when the main purpose is to transfer text, which is why mail transfer protocols, instant messaging, and news aggregation protocols are usually text based. Text protocols must have structures similar to binary protocols. The reason is that, although their main content differs, both share the goal of transferring data from one place to another.

The following section details some common text protocol structures that you'll likely encounter in the real world.

Numeric Data

Over the millennia, science and written languages have invented ways to represent numeric values in textual format. Of course, computer protocols don't need to be human readable, but why go out of your way just to prevent a protocol from being readable (unless your goal is deliberate obfuscation).

Integers

It's easy to represent integer values using the current character set's representation of the characters 0 through 9 (or A through F if hexadecimal). In this simple representation, size limitations are of no concern, and if a number needs to be larger than a binary word size, you can add digits. Of course, you'd better hope that the protocol parser can handle the extra digit or security issues will inevitably occur.

To make a signed number, you add the minus (–) character to the front of the number; the plus (+) symbol for positive numbers is implied.

Decimal Numbers

Decimal numbers are usually defined using human-readable forms. For example, you might write a number as 1.234, using the dot character to separate the integer and fractional components of the number; however, you'll still need to consider the requirement of parsing a value afterward.

Binary representations, such as floating point, can't represent all decimal values precisely with finite precision (just as decimals can't represent numbers like 1/3). This fact can make some values difficult to represent in text format and can cause security issues, especially when values are compared to one another.

Text Booleans

Booleans are easy to represent in text protocols. Usually, they're represented using the words *true* or *false*. But just to be difficult, some protocols might require that words be capitalized exactly to be valid. And sometimes integer values will be used instead of words, such as 0 for false and 1 for true, but not very often.

Dates and Times

At a simple level, it's easy to encode dates and times: just represent them as they would be written in a human-readable language. As long as all applications agree on the representation, that should suffice.

Unfortunately, not everyone can agree on a standard format, so typically many competing date representations are in use. This can be a particularly acute issue in applications such as mail clients, which need to process all manner of international date formats.

Variable-Length Data

All but the most trivial protocols must have a way to separate important text fields so they can be easily interpreted. When a text field is separated out of the original protocol, it's commonly referred to as a *token*. Some protocols specify a fixed length for tokens, but it's far more common to require some type of variable-length data.

Delimited Text

Separating tokens by delimiting characters is a very common way to separate tokens and fields that's simple to understand and easy to construct and parse. Any character can be used as the delimiter (depending on the type of data being transferred), but whitespace is encountered most in human-readable formats. That said, the delimiter doesn't have to be whitespace. For example, the Financial Information Exchange (FIX) protocol delimits tokens using the ASCII Start of Header (SOH) character with a value of 1.

Terminated Text

Protocols that specify a way to separate individual tokens must also have a way to define an End of Command condition. If a protocol is broken into separate lines, the lines must be terminated in some way. Most well-known, text-based Internet protocols are *line oriented*, such as HTTP and IRC; lines typically delimit entire structures, such as the end of a command.

What constitutes the end-of-line character? That depends on whom you ask. OS developers usually define the end-of-line character as either the ASCII *Line Feed (LF)*, which has the value 10; the *Carriage Return (CR)* with the value 13; or the combination CR LF. Protocols such as HTTP and Simple Mail Transfer Protocol (SMTP) specify CR LF as the official end-of-line combination. However, so many incorrect implementations occur that most parsers will also accept a bare LF as the end-of-line indication.

Structured Text Formats

As with structured formats such ASN.1, there is normally no reason to reinvent the wheel when you want to represent structured data in a text protocol. You might think of structured text formats as delimited text on steroids, and as such, rules must be in place for how values are represented and hierarchies constructed. With this in mind, I'll describe three formats in common use within real-world text protocols.

Multipurpose Internet Mail Extensions

Originally developed for sending multipart email messages, *Multipurpose Internet Mail Extensions (MIME)* found its way into a number of protocols, such as HTTP. The specification in RFCs 2045, 2046 and 2047, along with numerous other related RFCs, defines a way of encoding multiple discrete attachments in a single MIME-encoded message.

MIME messages separate the body parts by defining a common separator line prefixed with two dashes (--). The message is terminated by following this separator with the same two dashes. Listing 3-3 shows an example of a text message combined with a binary version of the same message.

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=MSG_2934894829

This is a message with multiple parts in MIME format.
--MSG_2934894829
Content-Type: text/plain

Hello World!
--MSG_2934894829
Content-Type: application/octet-stream
Content-Transfer-Encoding: base64

PGh0bWw+Cjxib2R5PgpIZWxsbyBXb3JsZCEKPC9ib2R5Pgo8L2h0bWw+Cg==
--MSG_2934894829--
```

Listing 3-3: Simple MIME message

One of the most common uses of MIME is for Content-Type values, which are usually referred to as *MIME types*. A MIME type is widely used when serving HTTP content and in operating systems to map an application to a particular content type. Each type consists of the form of the data it represents, such as *text* or *application*, in the format of the data. In this case, `plain` is unencoded text and `octet-stream` is a series of bytes.

JavaScript Object Notation

JavaScript Object Notation (JSON) was designed as a simple representation for a structure based on the object format provided by the JavaScript programming language. It was originally used to transfer data between a web page in a browser and a backend service, such as in Asynchronous JavaScript and XML (AJAX). Currently, it's commonly used for web service data transfer and all manner of other protocols.

The JSON format is simple: a JSON object is enclosed using the braces ({}) ASCII characters. Within these braces are zero or more member entries, each consisting of a key and a value. For example, Listing 3-4 shows a simple JSON object consisting of an integer index value, "Hello world!" as a string, and an array of strings.

```
{
    "index" : 0,
    "str" : "Hello World!",
    "arr" : [ "A", "B" ]
}
```

Listing 3-4: Simple JSON object

The JSON format was designed for JavaScript processing, and it can be parsed using the `"eval"` function. Unfortunately, using this function comes with a significant security risk; namely, it's possible to insert arbitrary script code during object creation. Although most modern applications use a parsing library that doesn't need a connection to JavaScript, it's worth ensuring that arbitrary JavaScript code is not executed in the context of the application. The reason is that it could lead to potential security issues, such as *cross-site scripting (XSS)*, a vulnerability where attacker-controlled JavaScript can be executed in the context of another web page, allowing the attacker to access the page's secure resources.

Extensible Markup Language

Extensible Markup Language (XML) is a markup language for describing a structured document format. Developed by the W3C, it's derived from Standard Generalized Markup Language (SGML). It has many similarities to HTML, but it aims to be stricter in its definition in order to simplify parsers and create fewer security issues.[1]

At a basic level, XML consists of elements, attributes, and text. *Elements* are the main structural values. They have a name and can contain child elements or text content. Only one root element is allowed in a single document. *Attributes* are additional name-value pairs that can be assigned to an element. They take the form of *`name="Value"`*. Text content is just that, text. Text is a child of an element or the value component of an attribute.

Listing 3-5 shows a very simple XML document with elements, attributes, and text values.

```
<value index="0">    <str>Hello World!</str>
    <arr><value>A</value><value>B</value></arr>
</value>
```

Listing 3-5: Simple XML document

All XML data is text; no type information is provided for in the XML specification, so the parser must know what the values represent. Certain specifications, such as XML Schema, aim to remedy this type information deficiency but they are not required in order to process XML content. The XML specification defines a list of well-formed criteria that can be used to determine whether an XML document meets a minimal level of structure.

XML is used in many different places to define the way information is transmitted in a protocol, such as in Rich Site Summary (RSS). It can also be part of a protocol, as in Extensible Messaging and Presence Protocol (XMPP).

[1] Just ask those who have tried to parse HTML for errant script code how difficult that task can be without a strict format.

Encoding Binary Data

In the early history of computer communication, 8-bit bytes were not the norm. Because most communication was text based and focused on English-speaking countries, it made economic sense to send only 7 bits per byte as required by the ASCII standard. This allowed other bits to provide control for serial link protocols or to improve performance. This history is reflected heavily in some early network protocols, such as the SMTP or Network News Transfer Protocol (NNTP), which assume 7-bit communication channels.

But a 7-bit limitation presents a problem if you want to send that amusing picture to your friend via email or you want to write your mail in a non-English character set. To overcome this limitation, developers devised a number of ways to encode binary data as text, each with varying degrees of efficiency or complexity.

As it turns out, the ability to convert binary content into text still has its advantages. For example, if you wanted to send binary data in a structured text format, such as JSON or XML, you might need to ensure that delimiters were appropriately escaped. Instead, you can choose an existing encoding format, such as Base64, to send the binary data and it will be easily understood on both sides.

Let's look at some of the more common binary-to-text encoding schemes you're likely to encounter when inspecting a text protocol.

Hex Encoding

One of the most naive encoding techniques for binary data is *hex encoding*. In hex encoding, each octet is split into two 4-bit values that are converted to two text characters denoting the hexadecimal representation. The result is a simple representation of the binary in text form, as shown in Figure 3-18.

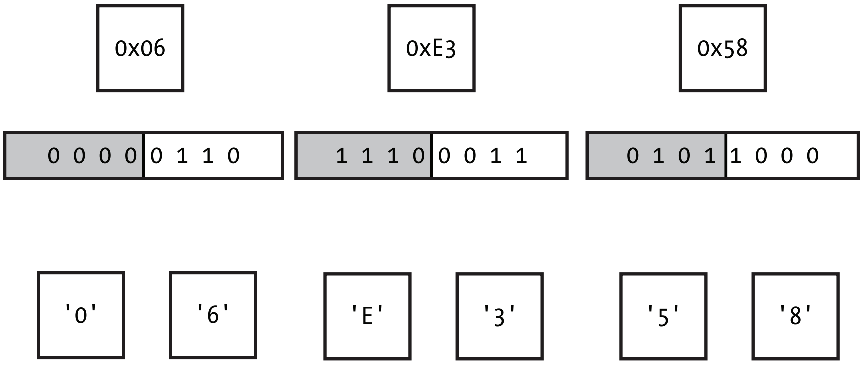


Figure 3-18: Example hex encoding of binary data

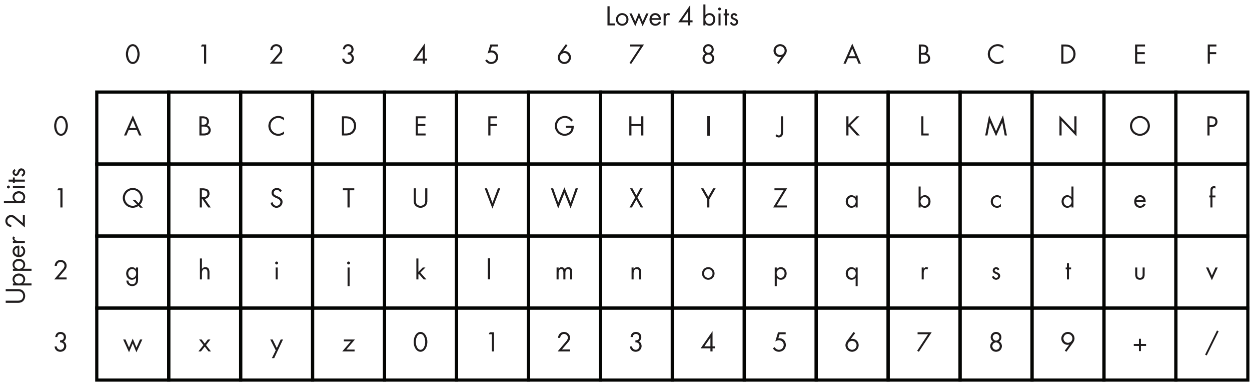
Although simple, hex encoding is not space efficient because all binary data automatically becomes 100 percent larger than it was originally. But one advantage is that encoding and decoding operations are fast and simple and little can go wrong, which is definitely beneficial from a security perspective.

HTTP specifies a similar encoding for URLs and some text protocols called *percent encoding*. Rather than all data being encoded, only nonprintable data is converted to hex, and values are signified by prefixing the value with a % character. If percent encoding was used to encode the value in Figure 3-18, you would get %06%E3%58.

Base64

To counter the obvious inefficiencies in hex encoding, we can use Base64, an encoding scheme originally developed as part of the MIME specifications. The *64* in the name refers to the number of characters used to encode the data. Think of Base64 as a number base, just as Base2 is binary and Base16 is hexadecimal. (Following that logic, hex encoding could be called Base16.)

The input binary is separated into individual six-bit values, enough to represent 0 through 63. This value is then used to look up a corresponding character in an encoding table, as shown in Figure 3-19.



Lower 4 bits

Upper 2 bits	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Q	R	S	T	U	V	W	X	Y	Z	a	b	c	d	e	f
2	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
3	w	x	y	z	0	1	2	3	4	5	6	7	8	9	+	/

Figure 3-19: Base64 encoding table

But there's a problem with this approach: when 8 bits are divided by 6, 2 bits remain. To counter this problem, the input is taken in units of three octets, because dividing 24 bits by 6 bits produces 4 values. Thus, Base64 encodes 3 bytes into 4, representing an increase of only 33 percent, which is significantly better than the increase produced by hex encoding. Figure 3-20 shows an example of encoding a three-octet sequence into Base64.

But yet another issue is apparent with this strategy. What if you have only one or two octets to encode? Would that not cause the encoding to fail? Base64 gets around this issue by defining a placeholder character, the equal sign (=). If in the encoding process, no valid bits are available to use, the encoder will encode that value as the placeholder. Figure 3-21 shows an example of only one octet being encoded. Note that it generates two placeholder characters. If two octets were encoded, Base64 would generate only one.

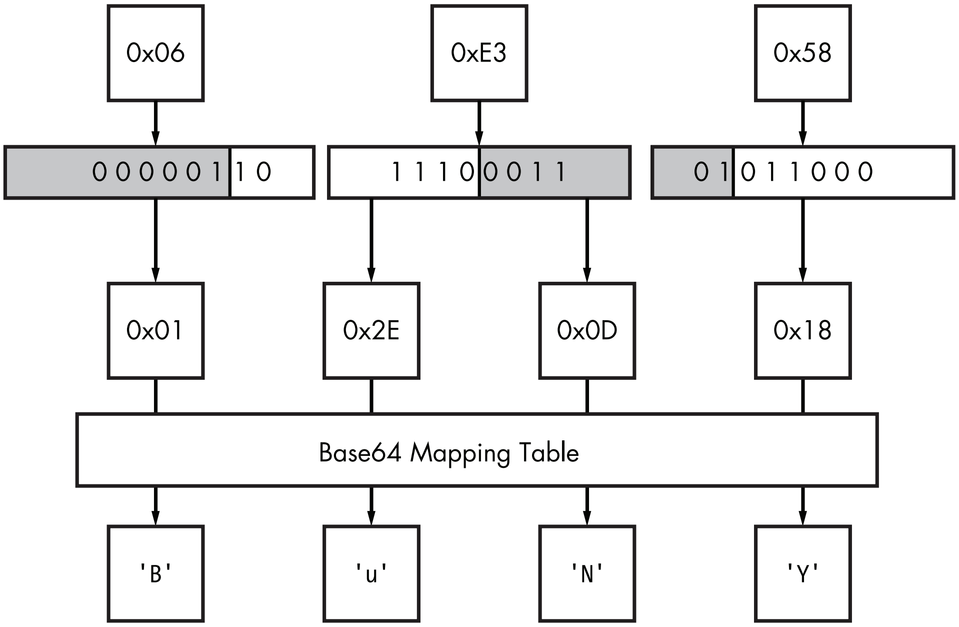


Figure 3-20: Base64 encoding 3 bytes as 4 characters

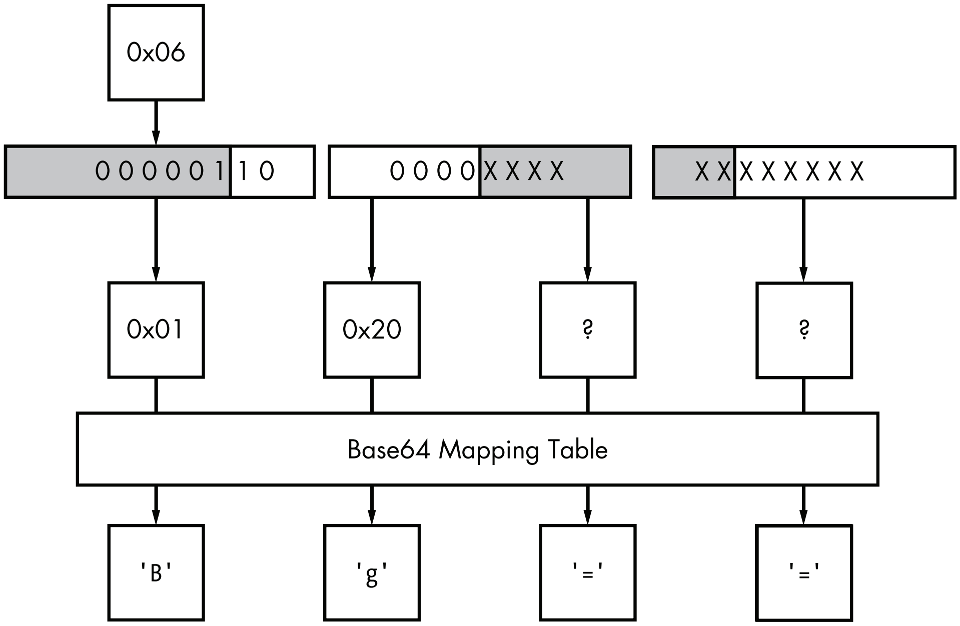


Figure 3-21: Base64 encoding 1 byte as 3 characters

To convert Base64 data back into binary, you simply follow the steps in reverse. But what happens when a non-Base64 character is encountered during the decoding? Well that's up to the application to decide. We can only hope that it makes a secure decision.

Final Words

In this chapter, I defined many ways to represent data values in binary and text protocols and discussed how to represent numeric data, such as integers, in binary. Understanding how octets are transmitted in a protocol is crucial to successfully decoding values. At the same time, it's also important to identify the many ways that variable-length data values can be represented because they are perhaps the most important structure you will encounter within a network protocol. As you analyze more network protocols, you'll see the same structures used repeatedly. Being able to quickly identify the structures is key to easily processing unknown protocols.

In Chapter 4, we'll look at a few real-world protocols and dissect them to see how they match up with the descriptions presented in this chapter.