Various Application Layer Protocols

In addition to DNS and HTTP, there are several other important application layer protocols. This section will cover a few of them.

TLS

The two most common protocols you see in web browsers are HTTP and HTTPs. The s in HTTPs stands for "secure". A big problem with plain HTTP is that it is not encrypted, so usernames, passwords, and everything else are visible to anyone intercepting HTTP traffic.

The security in HTTPs is provided by a protocol called TLS (*Transport Layer Security*). People use the term SSL (*Secure Sockets Layer*) interchangeably with TLS. SSL came first, and SSL version 3 was renamed TLS. But often people still use the older term.

TLS provides three things: *encryption*, *integrity*, and *authentication*. Encryption is where the contents of a message are scrambled so that they are unreadable to anyone but the intended recipient. Integrity is where the recipient can be sure that the message they receive was not tampered with. Even if a message is encrypted, someone intercepting a message can remove or modify the some of the encrypted text, and those modifications can in turn affect what it decrypts back to. Integrity provides a way to determine if a modification has happened. Authentication is the process where a client can be sure that the server really is who it claims to be and not someone impersonating it.

Encryption and integrity and provided by some pretty secure mathematical algorithms. Authentication is the weak link because it relies on humans in the verification process. The process uses something called *certificates* which certify that someone is who they say they are. People and companies apply for certificates, and a company called a *certificate authority* is supposed to verify their documentation. But not all certificate authorities take this verification process seriously.

A TLS session starts with the client identifying itself to the server and indicating what types of encryption it supports. The server picks the strongest type of encryption from the client's list that it also supports. The server also sends a copy of its certificate for the client to verify. This whole process is called a *TLS handshake*. Once this is established, then ordinary HTTP information can be sent back and forth, but not it will be secured.

TLS encrypts all of the data of the encryption, but it does not currently encrypt other things such as the name of the website or IP addresses. Times and the amount of data transferred are also visible. All of this is known as *metadata*. It is information about the transaction itself, and often that information can be useful to people.

TLS has been very successful. The majority of web traffic is now HTTPs, especially since some web browsers show warning messages when accessing pages over plain HTTP. TLS is also used to secure email, and HTTPs is starting to be used to secure DNS queries.

Telnet and SSH

The main purpose of Telnet and SSH is to remotely log onto another computer and perform actions on that computer. Telnet is a very old protocol that has largely been superseded by SSH. The reason for this is simple: Telnet is not encrypted. Anything you send over Telnet, including your username and password, are visible to anyone that intercepts your network traffic.

Telnet for a long time was *the* way to connect remotely to other computers. It's still used to connect to some older machines that haven't been updated or in cases where encryption is not important. SSH is widely used by system administrators to remotely manage systems.

FTP

Before HTTP, if you wanted to transfer files on the internet, the *File Transfer Protocol* (FTP) was the tool of choice. It still works, though it's becoming rare. Just like Telnet, its major weakness is that it does not encrypt things, so usernames, passwords, and file contents are all visible to anyone intercepting traffic. There is a secure replacement, SFTP, that is commonly used to securely transfer files, manage directories, and do other basic operations. It uses SSH under the hood to secure the connection.

Email

Email dates back to the 1970s. There are several protocols associated with it. Email is managed by mail servers that store and exchange email. If `alice@msmary.edu` wants to send an email to `bob@example.com`, Alice talks to the mail server for `msmary.edu`. That mail server will then look up the mail server for `example.com` (via a `MX` DNS resource record) and send the mail over to that server. It might possibly pass through other mail servers on the way. Bob at some point will contact his mail server, see that Alice's message has arrived, and he will download it.

There are three major protocols in use here. The *Simple Mail Transfer Protocol* (SMTP) is used by Alice to submit her email to her mail server, and it's used by her mail server to send the email to other mail servers. When Bob downloads the email, he'll use either the *Post Office Protocol, version 3* (POP3) or the *Internet Message Access Protocol* (IMAP). Microsoft Exchange (what you use if you use Outlook) sometimes uses its own special protocol for downloading email.

Here is a short example of what SMTP looks like when Alice wants to submit her email to her mail server.

```
HELO
AUTH LOGIN
[username and password would go here]
MAIL FROM: <alice@msmary.edu>
RCPT TO: <bob@example.com>
DATA
I can't get into my email account!
QUIT
```

Alice would never type this in herself. Her email program does this for her, but she could in theory do it herself, and it's a fun exercise to try.

Many email programs have an option where you can view the "original" of an email. This will show you a variety of headers that are sent along with the message from mail server to mail server. One of the most useful headers is the Received header. There may be multiple of these in the email, and each indicates a mail server that the message passed through on its way to its destination.

Of the two retrieval protocols, POP3 is the simpler one. Here is an example of what it looks like.

```
LIST
RETR 1
DELE 1
RETR 2
DELE 2
```

The LIST command shows what emails are on the server, and the other commands retrieve (download) and delete the mail from the server. IMAP is more sophisticated. It allows for multiple mailboxes and folders on a system. Many email applications are web-based and hide all of this from you. But if you want to use your own email client rather than, say, the standard Gmail program, you can do so using POP3 or IMAP.

SMTP, POP3, and IMAP are not encrypted. Any usernames and passwords, as well as the entire contents of the email, are visible to anyone sniffing network traffic. Because of this, SMTP, POP3, and IMAP are all often run over TLS in a similar way to how HTTP is run over TLS to create HTTPs.

Encodings and Base64

SMTP is old, and it has a serious limitation: all of the data must be in 7-bit ASCII. What does this mean? Characters, like everything else on a computer, are stored in binary. The type of *encoding* tells how to interpret that binary data. One of the earliest encodings still in common use is 7-bit ASCII. In it, each character is associated with a code from 0 to 127 (0000000 to 1111111 in binary). For instance, the letters A has code 65 (1000001 in binary). Capital letters occupy the range from 65 to 90. Lowercase letters occupy 97 to 122, and digits occupy the range from 48 to 57. Various other symbols occupy the rest of the values, with many of the values being used for control characters on teletypes, many of which are not relevant anymore.

In short, 7-bit ASCII can represent all of the characters on a standard American keyboard, and not much else. Letters with umlauts, like ü, Greek letters, Cyrillic letters, and so much else does not fit into the 128-character 7-bit ASCII standard. There is a newer standard, called Unicode, that includes these and much more.

But if you want to send an email that contains any type of special character or contains a non-text attachment, like an image or a zip file, then something needs to be done since SMTP can only handle 7-bit ASCII characters. The solution is something called *Base64 encoding*. It is a way to take arbitrary binary data and represent it in 7-bit ASCII. It's used in email and in many other applications.

Number systems

Base64 is actually a number system similar to binary or decimal. Though you may have seen this elsewhere, let's review a little about number systems. Here are the most common ones in use in computer science:

Name	Base	Symbols
Binary	2	0, 1
Octal	8	0–7
Decimal	10	0–9
Hexadecimal	16	0–9, A–F
Base64	64	A-F, a-f, 0-9, +, /

Decimal is the number system we use in real life. It's called a base 10 system because we use 10 digits. Binary is what computers natively work in. Octal and hexadecimal (hex) are used because binary is long and painful to look at, and conversions between binary and decimal are painful.

Converting decimal to binary Let's look at an example of converting the number 43 from decimal to binary. Start by laying out slots with powers of 2 like below on the left. Then we will fill in the slots, eventually ending up with the answer on the right.

___	___	___	___	___	___	___	___
128	64	32	16	8	4	2	1

0	0	1	0	0	1	1	1
128	64	32	16	8	4	2	1

Start with 128 and see if we can fit any 128's into 43. We can't, so we mark the 128's slot with 0 and move on to 64. Again, we can't fit any 64's into 43, so we mark the 64's slot with a 0 and move on to 32. We can fit a 32 into 43, so we mark the 32's slot with a 1. We then do $43 - 32 = 11$, and 11 is what we now work with.

We check to see if we can fit any 16's into 11. We can't, so we mark the 16's slot with 0 and move onto 8. We can fit an 8 into 11, so we mark the 8's slot with 1 and do $11 - 8 = 3$. Now we work with 3. We see if we can fit any 4's into 3, but we can't, so we mark the 4's slot with 0. Then we see if we can fit any 2's into 3, which we can, so we mark the 2's slot with a 1. Then we do $3 - 2 = 1$, and we work with 1. Finally, we see if any 1's fit into 1, which they do, so we mark the 1's slot with a 1, and we're done. The final binary number is 00101011 or 101011 if we leave off the initial zeroes.

To convert from binary to decimal, we can do the process in reverse. Take the number 101011 that we just computed. Draw it out with all the powers of two below the slots like above. There are 1's in the slots for 32, 8, 2, and 1, so the conversion is $32 + 8 + 2 + 1 = 43$.

Converting with hexadecimal Converting between binary and decimal is a pain, so very early on, people decided to start working with hexadecimal. The reason is that because 2 and 16 are both powers of 2, the conversion turns out to be quick. Specifically, because $2^4 = 16$, we can convert binary to hex by combining the binary digits into groups of 4s. The table below will help with that. It shows decimal, hex, and binary for the values 0 to 15. If you do enough of this, the conversions start to become second nature.

0	0	0000	4	4	0100	8	8	1000	12	C	1100
1	1	0001	5	5	0101	9	9	1001	13	D	1101
2	2	0010	6	6	0110	10	A	1010	14	E	1110
3	3	0011	7	7	0111	11	B	1011	15	F	1111

Here is how to use the table. Suppose we want to convert 0100001111011111 to hex. We start by breaking it into groups of four, and then convert the groups using the table above. The result is 83DF, which is often written in the form `0x83DF`, where the `0x` indicates a hex number.

0100	0011	1101	1111
8	3	D	F

Going from hex to binary is the same idea but in reverse.

Converting with Base64 Converting binary to Base64 is similar. Since 64 is also a power of 2, namely $2^6 = 64$, we can do the conversion by grouping. Here the groups are of 6 binary bits. Also, remember that the Base64 system is a little different from hex in that instead of running 0-9 and A-F, it starts with letters, so that A corresponds to 0, B to 1, etc. The order is A–Z, a-z, 0-9, +, /.

Let's convert the binary number 011000010110001001100011' to Base64. We break it into groups of 6 as below. The decimal equivalents are shown below that and at the bottom is what they translate to in Base64.

011000	010110	001001	100011
24	22	9	35
Y	W	J	j

So the result is YWJj. Note that depending on the length of the number being encoded, for certain technical reasons, you may see = or == at the end of the Base64 representation.

There are many sites online that you can use to do Base64 encodings and decodings. Python's built-in `base64` module can do it also. Here are two quick examples of encoding and decoding.

```
from base64 import b64encode, b64decode

print(b64encode(b'abc'))
print(b64decode('YWJj'))
```

Note the `b` in front of the string in the encoding example. That is to indicate it's not a regular string, but a byte representation of a string. We need to this because Python's `b64encode` won't work with ordinary strings. You could also transform a string called `s` into bytes by doing `s.encode('utf-8')`

Finally, so we can see this in action, here is some text I copied from a raw email. We see that the entire contents of the file are encoded in Base64.

```
Content-Type: text/plain; name="rhythm.txt"
Content-Description: rhythm.txt
Content-Disposition: attachment; filename="rhythm.txt"; size=258;
creation-date="Tue, 01 Sep 2020 12:14:48 GMT";
modification-date="Tue, 01 Sep 2020 12:14:48 GMT"
Content-Transfer-Encoding: base64
```

ZnJvbSB3aW5zb3VuZCBpbXBvcnQgQmVlcAOKZnJvbSBOaW1lIGltcG9ydCBzbGVlcAOKDQpuIDOg
MgOKbSA9IDUNCgOKZm9yIGkgaW4gcmFuZ2UoMTAwKToNCiAgICBpZiBpICUgKGOqbikgPTOgMDoN
CiAgICAgICAgICAgIEJlZXAoMjAwMCwgNTApDQogICAgZWxpZiBpICUgbiA9PSAwOgOKICAgICAg
ICAgICAgQmVlcCgzMDAwLCA1MCkNCiAgICBlbGlmIGkgJSBtID09IDA6DQogICAgICAgICAgICBC
ZWVwKDEwMDAsIDUwKQOKICAgIHNsZWVwKC4xKQOK