EECS 440 System Design of a Search Engine

Winter 2021

Lecture 5: TCP/IP DNS and Sockets

Nicole Hamilton
https://web.eecs.umich.edu/~nham/
nham@umich.edu

Agenda

1. Course details.
2. Objective: Read a webpage.
3. TCP/IP.
4. DNS.
5. Sockets.
6. LinuxGetUrl
7. LinuxGetSsl

Agenda

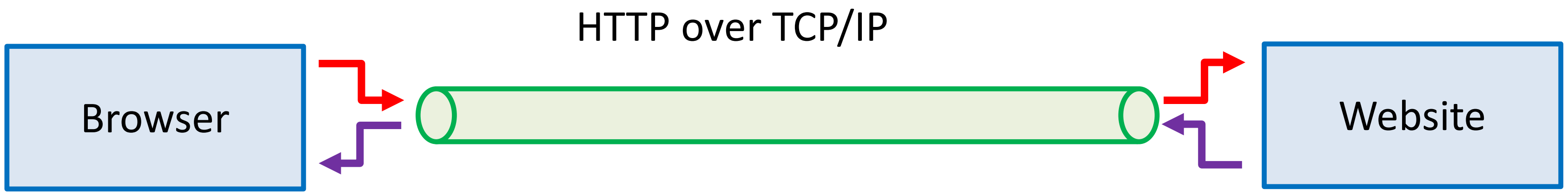
1. Course details.
2. Objective: Read a webpage.
3. TCP/IP.
4. DNS.
5. Sockets.
6. LinuxGetUrl

details

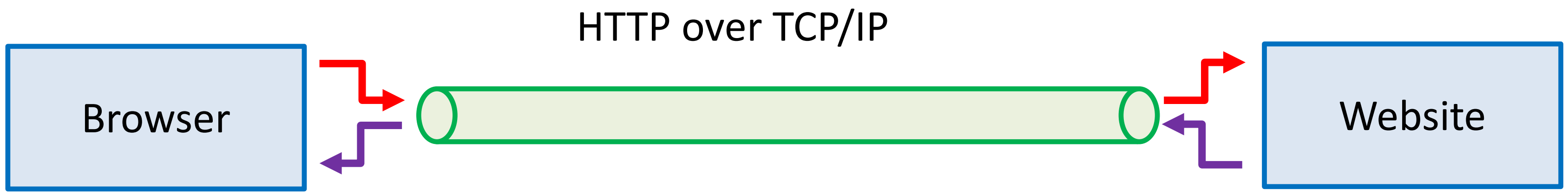
1. Anyone still looking for a group?
2. Tee shirts?
3. Group photo due Sep 23 and I’ll want to meet with each group shortly after.
4. Project plans due Oct 6.
5. Still fiddling with the schedule.

Agenda

1. Course details.
2. Objective: Read a webpage.
3. TCP/IP.
4. DNS.
5. Sockets.
6. LinuxGetUrl



Imagine the connection between a browser and a website as a long pipe.
At each end is a socket you can read or write from as if it was a file.
Anything written into one end pops out and can be read at the other.



To read a page from a website:

1. Look up the TCP/IP address of the website.
2. Create a socket.
3. Connect the socket to that address.
4. Send a GET message to request the page.
5. Read what comes back.

Reading and serving webpages

We’ll discuss what’s needed to build the first of three small projects we’re planning as lab and hw exercises:

1.	LinuxGetUrl	Read an HTTP page.
2.	LinuxGetSsl	Read an HTTPS page.
3.	LinuxTinyServer	A simple HTTP server.

Here’s LinuxTinyServer.

```
tcsh-3% head LinuxTinyServer.cpp
// Linux tiny HTTP server.
// Nicole Hamilton  nham@umich.edu

// This variation of LinuxTinyServer supports a simple plugin interface
// to allow "magic paths" to be intercepted.

// Usage:  LinuxTinyServer port rootdirectory

// Compile with g++ -pthread LinuxTinyServer.cpp -o LinuxTinyServer
// To run under WSL (Windows Subsystem for Linux), must elevate with
tcsh-4% ls website
Images  Styles  index.htm
tcsh-5% ./LinuxTinyServer 5000 website
Listening on 0.0.0.0:5000
```

LinuxGetUrl does an HTTP Get.

```
tcsh-5% head LinuxGetUrl.cpp
// Linux get URL utility that copies the HTTP page to stdout.
// Nicole Hamilton  nham@umich.edu

#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <iostream>
#include <string.h>
#include <string>
tcsh-6% LinuxGetUrl
Usage:  LinuxGetUrl url
tcsh-7% ./LinuxGetUrl http://localhost:5000/index.htm | head -20
Service = http, Host = localhost, Port = 5000, Path = index.htm
Host address length = 16 bytes
Family = 2, port = 5000, address = 127.0.0.1
GET /index.htm HTTP/1.1
Host: localhost
User-Agent: LinuxGetUrl/2.0 nham@umich.edu (Linux)
Accept: */*
Accept-Encoding: identity
Connection: close
```

The server sees the Get request and returns the file.

```
tcsh-5% ./LinuxTinyServer 5000 website
Listening on 0.0.0.0:5000

Connection accepted from 127.0.0.1:56032

GET /index.htm HTTP/1.1
Host: localhost
User-Agent: LinuxGetUrl/2.0 nham@umich.edu (Linux)
Accept: */*
Accept-Encoding: identity
Connection: close

Requested path = /index.htm
Actual path = website/index.htm

HTTP/1.1 200 OK
Content-Length: 8964
Connection: close
Content-Type: text/html
```

LinuxGetUrl reads the file.

```
tcsh-7% ./LinuxGetUrl http://localhost:5000/index.htm | head -20
Service = http, Host = localhost, Port = 5000, Path = index.htm
Host address length = 16 bytes
Family = 2, port = 5000, address = 127.0.0.1
GET /index.htm HTTP/1.1
Host: localhost
User-Agent: LinuxGetUrl/2.0 nham@umich.edu (Linux)
Accept: */*
Accept-Encoding: identity
Connection: close


HTTP/1.1 200 OK
Content-Length: 8964
Connection: close
Content-Type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

tcsh-8%
```

Errors are reported 400 and other codes.

```
tcsh-2% ./LinuxGetUrl http://localhost:5000/zork.htm
Service = http, Host = localhost, Port = 5000, Path = zork.htm
Host address length = 16 bytes
Family = 2, port = 5000, address = 127.0.0.1
GET /zork.htm HTTP/1.1
Host: localhost
User-Agent: LinuxGetUrl/2.0 nham@umich.edu (Linux)
Accept: */*
Accept-Encoding: identity
Connection: close


HTTP/1.1 404 Not Found
Content-Length: 0
Connection: close

tcsh-3%
```

We’ll now go through the mechanics of making this happen.

Agenda

1. Course details.
2. Objective: Read a webpage.
3. TCP/IP.
4. DNS.
5. Sockets.
6. LinuxGetUrl

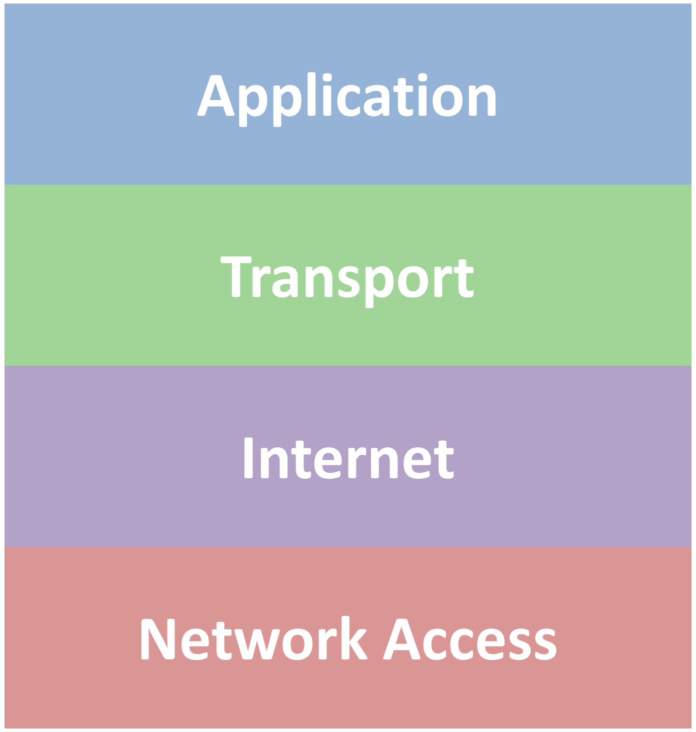
TCP/IP Model

DHCP, DNS, FTP, HTTP, HTTPS,
POP, SMTP, SSH, etc.

TCP and UDP

IP address: IPv4 or IPv6

Link level: MAC address
Physical: Cable, fiber, wireless



IP routing

Uses a routing table to select a next hop router.

Given a destination IP address, **D**, and network prefix, **N**:

```
if ( N matches a directly connected network address )
   Deliver datagram to D over that network link;
else if ( The routing table contains a route for N )
   Send datagram to the next-hop address listed in the routing table;
else if ( a default route exists )
   Send datagram to the default route;
else
   Send a forwarding error message to the originator;
```

Agenda

1. Course details.
2. Objective: Read a webpage.
3. TCP/IP.
4. DNS.
5. Sockets.
6. LinuxGetUrl

To get an IP address

1. Parse the HTTPS path to identify the host (domain name) we’re trying to reach.
2. Find the IP address for that host using a Domain Name Server (DNS).

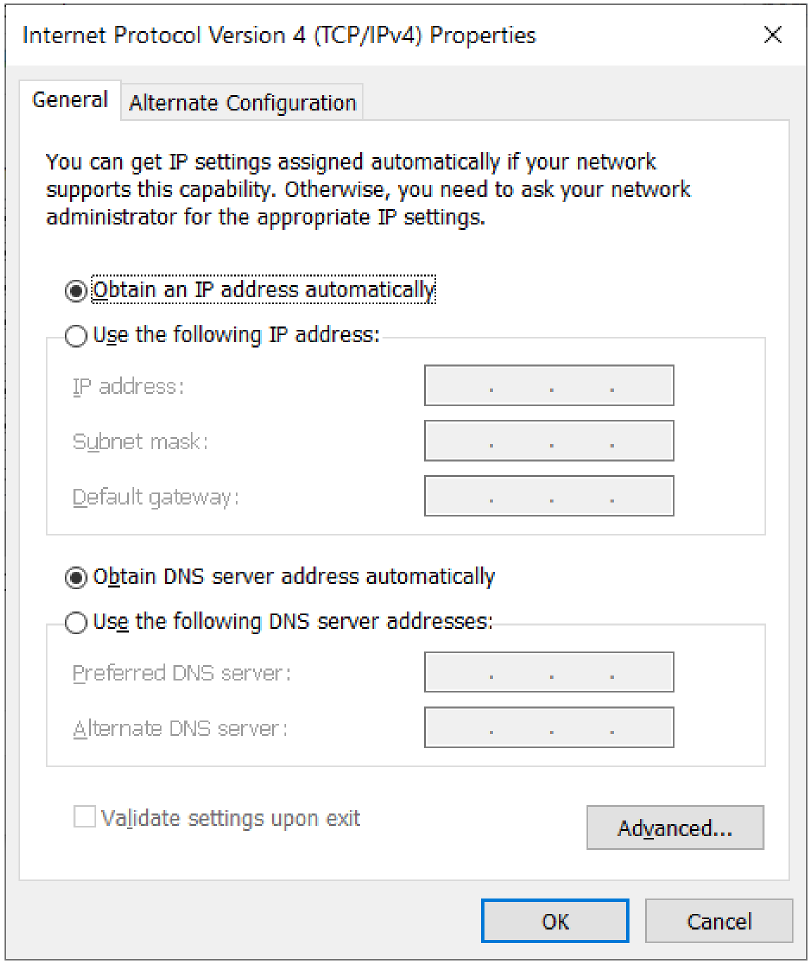
DNS Records

```
Type      Name      Value     TTL       Actions
A         @         160.153.46.5        600 seconds
A         admin     160.153.46.5        600 seconds
A         mail      160.153.46.5        600 seconds
:
CNAME     webmail   @         1 Hour
CNAME     www       @         1 Hour
:
MX        @         mail.hamiltonlabs.com (Priority: 0)  1 Hour    Edit
NS        @         ns61.domaincontrol.com      1 Hour
NS        @         ns62.domaincontrol.com      1 Hour
SOA       @         Primary nameserver: ns61.domaincontrol.com.    600 seconds
```

An A record defines a host address.
A CNAME record defines a canonical name for alias.
An MX (Mail eXchange) record defines a mail server.
An NS record defines a name server.
An SOA (Start of Authority) defines the primary name server.

DHCP

We usually rely on DHCP (Dynamic Host Configuration Protocol) to assign an IP address to our machine and DNS server.



Internet Protocol Version 4 (TCP/IPv4) Properties

General | Alternate Configuration

You can get IP settings assigned automatically if your network supports this capability. Otherwise, you need to ask your network administrator for the appropriate IP settings.

[✓ Obtain an IP address automatically]
Use the following IP address:

IP address:	. . .
Subnet mask:	. . .
Default gateway:	. . .

[✓ Obtain DNS server address automatically]
Use the following DNS server addresses:

Preferred DNS server:	. . .
Alternate DNS server:	. . .

☐ Validate settings upon exit

Let’s assume a simple mechanism for parsing a full URL into the components.

```
class ParsedUrl
   {
   public:
      const char  *CompleteUrl;
      char        *Service,
                  *Host,
                  *Port,
                  *Path;

      ParsedUrl( const char *url );
      ~ParsedUrl( );
   };
```

Example use:

```
ParsedUrl url( "http://localhost:5000/index.htm" );
cout << "Service = " << url.Service <<
   ", Host = " << url.Host <<
   ", Port = " << url.Port <<
   ", Path = " << url.Path << endl;
```

Should print:

```
Service = http, Host = localhost, Port = 5000, Path = index.htm
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *node, const char *service,
               const struct addrinfo *hints,
               struct addrinfo **res);

void freeaddrinfo(struct addrinfo *res);
```

Given node and service, which identify an Internet host domain and a service, getaddrinfo() returns one or more addrinfo structures, each of which contains an Internet address that can be specified in a call to bind(2) or connect(2).

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *node, const char *service,
               const struct addrinfo *hints,
               struct addrinfo **res);

void freeaddrinfo(struct addrinfo *res);
```

Given node and service, which identify an Internet host and a service, getaddrinfo() returns one or more addrinfo structures, each of which contains an Internet address that can be specified in a call to bind(2) or connect(2) to that website.

Here's an example use.

```
// Get the host address, supplying hints for
// what we're looking for.

struct addrinfo *address, hints;
memset( &hints, 0, sizeof( hints ) );
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;

int getaddrResult = getaddrinfo( url.Host,
      *url.Port ? url.Port : "80", &hints, &address );
```

Later, it must be freed.

```
freeaddrinfo( address );
```

This is what the addrinfo structure looks like. It contains an Internet address that can be specified in a call to bind(2) or connect(2).

```
struct addrinfo {
   int              ai_flags;
   int              ai_family;
   int              ai_socktype;
   int              ai_protocol;
   socklen_t        ai_addrlen;
   struct sockaddr *ai_addr;
   char            *ai_canonname;
   struct addrinfo *ai_next;
};
```

The interesting part is the ai_addr, the actual IP address, which we can print.

```
 PrintAddress( ( sockaddr_in * )address->ai_addr,
      sizeof( struct sockaddr ) );
```

Here’s a simple print routine.

```
void PrintAddress( const sockaddr_in *s, const size_t saLength )
   {
   const struct in_addr *ip = &s->sin_addr;
   uint32_t a = ntohl( ip->s_addr );

   cout << "Host address length = " << saLength << " bytes" << endl;
   cout << "Family = " << s->sin_family <<
         ", port = " << ntohs( s->sin_port ) <<
         ", address = " << ( a >> 24 ) << '.' <<
               ( ( a >> 16 ) & 0xff ) << '.' <<
               ( ( a >> 8 ) & 0xff ) << '.' <<
               ( a & 0xff ) << endl;
   }
```

Example use:

```
int getaddrResult = getaddrinfo( "www.nytimes.com", "80",
   &hints, &address );

PrintAddress( ( sockaddr_in * )address->ai_addr,
   sizeof( struct sockaddr ) );
```

Should print:

```
Host address length = 16 bytes
Family = 2, port = 80, address = 151.101.185.164
```

Agenda

1. Course details.
2. Objective: Read a webpage.
3. TCP/IP.
4. DNS.
5. Sockets.
6. LinuxGetUrl

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
int close(int fd);
```

socket() creates an endpoint for communication and returns a file descriptor that can be used for reading and writing.

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
int close(int fd);
```

The domain argument specifies a communication domain. Here are the most common:

Name	Purpose
AF_UNIX, AF_LOCAL	Local communication
AF_INET	IPv4 Internet protocols
AF_INET6	IPv6 Internet protocols

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
int close(int fd);
```

The socket has the indicated type, which specifies the communication semantics. The most common is SOCK_STREAM, a sequenced, reliable connection with two-way byte streams.

The protocol is usually IPPROTO_TCP.

```c
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

connect() connects the socket to the specified IP address. The addrlen argument specifies the size of addr structure.

Here’s an example creating a socket and connecting it to an address.

```c
// Create a TCP/IP socket.

int s = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );
assert( s != -1 );

// Connect the socket to the host address.

int connectResult = connect( s, address->ai_addr,
      sizeof( struct sockaddr ) );
assert( connectResult == 0 );
```

```c
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

send() writes data into the socket. recv() reads data. Flags allow close-on-exec, noblocking reads/writes and other options.

The only difference between send() and write() or between recv() and read() is the presence of the optional flags. With a zero flags argument, send() is equivalent to write() and recv() is equivalent to read().

Here’s a sample GET message we might send. Notice that the User-agent field must contain your contact info in 440.

```
GET / HTTP/1.1
Host: www.nytimes.com
User-Agent: LinuxGetUrl/2.0 nham@umich.edu (Linux)
Accept: */*
Accept-Encoding: identity
Connection: close
```

Some sites will not even respond without a User-Agent field. It’s a free text field and can be anything as long as it exists. It’s typically the name of the software product that generated the Get + a slash followed by a version number. The OS or build environment is usually given in parens.

The Accept: and Accept-Encoding: fields are not required but typically provided.

Here’s an example sending the Get message over socket s.

```
string getMessage;
:
send( s, getMessage.c_str( ), getMessage.length( ), 0 );
```

Here’s an example reading from a socket and writing to stdout.

```
char buffer[ 10240 ];
int bytes;

while ( ( bytes = recv( s, buffer, sizeof( buffer ), 0 ) ) > 0 )
   write( 1, buffer, bytes );
```

We’ll talk more about read( ) and write( ) when we get to the filesystem.

Agenda

1. Course details.
2. Objective: Read a webpage.
3. TCP/IP.
4. DNS.
5. Sockets.
6. LinuxGetUrl

Here’s the entire main( ), minus only all the code that you will have to write.

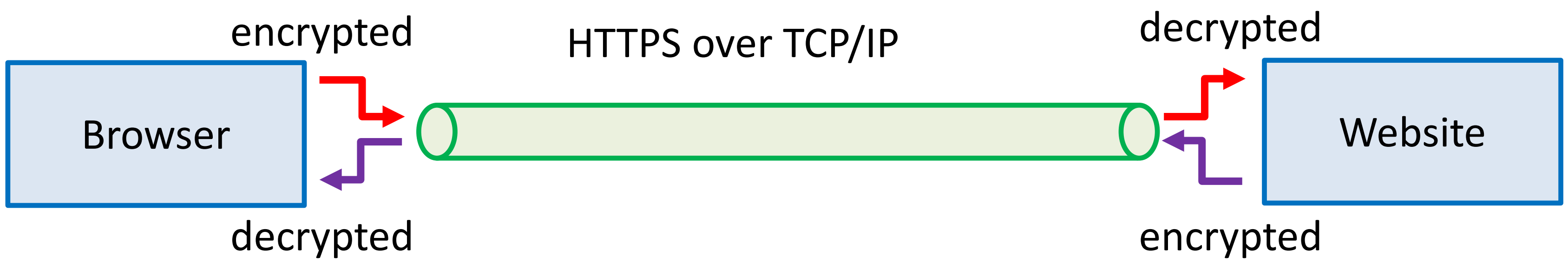
```
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int main( int argc, char **argv )
   {
   // Parse the URL

   // Get the host address.

   // Create a TCP/IP socket.

   // Connect the socket to the host address.

   // Send a GET message.

   // Read from the socket until there's no more data, copying it to
   // stdout.

   // Close the socket and free the address info structure.
   }
```

Agenda

1. Course details.
2. Objective: Read a webpage.
3. TCP/IP.
4. DNS.
5. Sockets.
6. LinuxGetUrl
7. Preview of SSL

Agenda

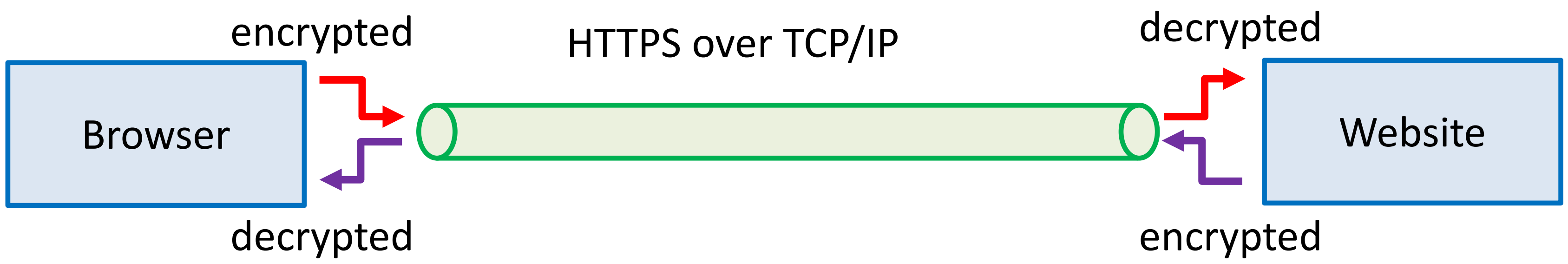
1. Course details.
2. Producer/consumer relationships and locks.
3. Sockets.
4. SSL.



Under HTTPS, data is encrypted before being sent and decrypted when received using a public key mechanism that allows both ends to agree on a secret session key.

Done using a Secure Socket Layer (SSL) wrapper around a regular socket.

Here, we'll use the OpenSSL library.



To read a page from a website:

1. Look up the TCP/IP address of the website.
2. Create a socket.
3. Connect the socket to that address.
4. Build an SSL layer and establish a secure connection.
5. Send a GET message to request the page.
6. Read what comes back.

```c
#include <openssl/ssl.h>

int SSL_library_init(void);
SSL_CTX *SSL_CTX_new(const SSL_METHOD *method);
int SSL_set_fd(SSL *ssl, int fd);
int SSL_connect(SSL *ssl);
int SSL_read(SSL *ssl, void *buf, int num);
int SSL_write(SSL *ssl, const void *buf, int num);
```

SSL_library_init() initializes the SSL library.
SSL_CTX_new() creates a new SSL_CTX object as framework to establish TLS/SSL enabled connections.

SSL_set_fd() sets the file descriptor fd as the input/output facility for the TLS/SSL (encrypted) side of ssl. fd will typically be the socket file descriptor of a network connection.

SSL_connect() initiates the TLS/SSL handshake with a server.

SSL_write() writes num bytes from the buffer buf into the specified ssl connection.
SSL_read() tries to read num bytes from the specified ssl into the buffer buf.

Here’s the entire main( ) for LinuxGetSSL, minus only all the code that you will write.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int main( int argc, char **argv )
   {
   // Parse the URL

   // Get the host address.

   // Create a TCP/IP socket.

   // Connect the socket to the host address.
 // Build an SSL layer and set it to read/write   // to the socke
t we've connected.
   // Fill in the socket we'll be using with SSL using
SSL_set_fd.
   // Send a GET message over the secure socket.

   // Read from the secure socket until there's no more data, copying
it to
   // stdout.
```