

EECS 498 System Design of a Search Engine

Winter 2026

Lecture 11: SSL

Nicole Hamilton

[https://web.eecs.umich.edu/~nham/
nham@umich.edu](https://web.eecs.umich.edu/~nham/nham@umich.edu)

agenda

1. details
2. Google Cloud accounts
3. demo redux
4. review of reading an HTTP webpage
5. read an HTTPS webpage
6. TLS, SSL and the OpenSSL library
7. LinuxGetSsl

agenda

1. details
2. midterm
3. Google Cloud accounts
4. demo redux
5. review of reading an HTTP webpage
6. read an HTTPS webpage
7. TLS, SSL and the OpenSSL library
8. LinuxGetSsl

details

1. A few of you have missed simple assignments, like voting in the survey. Talk to me about a makeup for partial credit.

agenda

1. details
2. midterm
3. Google Cloud accounts
4. demo redux
5. review of reading an HTTP webpage
6. read an HTTPS webpage
7. TLS, SSL and the OpenSSL library
8. LinuxGetSsl

midterm

1. Midterm is Thursday Feb 26 during lecture period.
2. 80 minutes, in person, on paper.
3. No notes or electronic devices.
4. W21 and W25 midterm solutions and a past review slide deck are posted in the Files > Exams > Midterm folder on Canvas.

agenda

1. details
2. midterm
3. Google Cloud accounts
4. demo redux
5. review of reading an HTTP webpage
6. read an HTTPS webpage
7. TLS, SSL and the OpenSSL library
8. LinuxGetSsl

Google Cloud accounts

1. Google Cloud accounts have been approved.
2. Link is posted on EdStem.
3. Make sure you can redeem your coupon.
4. We're planning some labs to help you deploy to the cloud when everyone's more ready to do that.
5. You only get \$50 each, so don't burn it up until the end, when you're preparing to demo.

agenda

1. details
2. midterm
3. Google Cloud accounts
4. demo redux
5. review of reading an HTTP webpage
6. read an HTTPS webpage
7. TLS, SSL and the OpenSSL library
8. LinuxGetSsl

localhost

Last time, I had problems. I could reach my server as localhost but not using the IP address.

But some of you were able to reach it using the IP address.

What gives?

Turns out this is by design, apparently to close off a possible exploit that would allow an attacker to get inside your firewall.

localhost

Here's the rule:

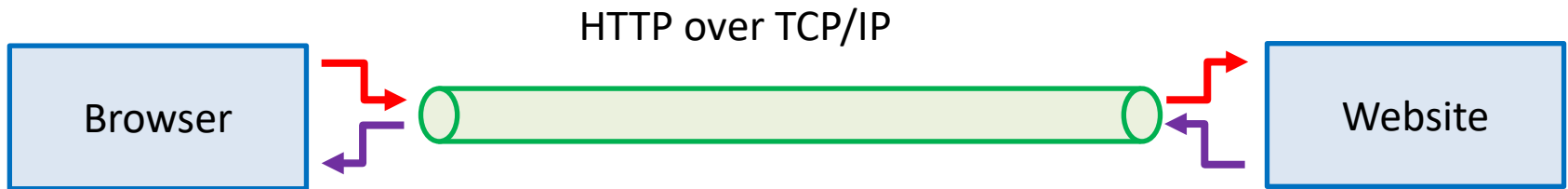
Use localhost to reach a server on your machine.

Use the IP address to reach a server on another machine.

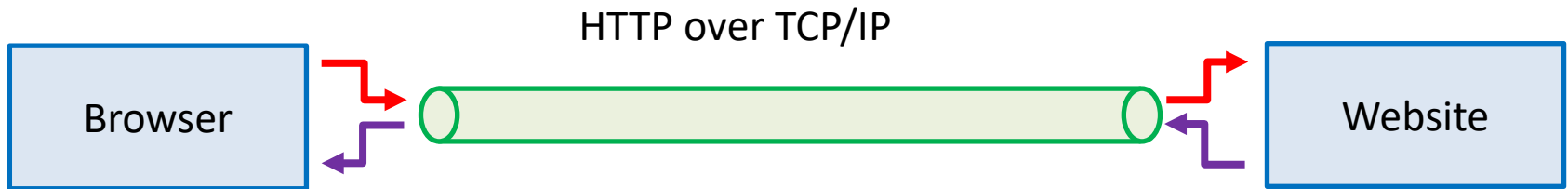
I'll now try to redo the demos.

agenda

1. details
2. Host: parameter
3. Review of reading an HTTP webpage
4. Read an HTTPS webpage
5. TLS, SSL and the OpenSSL library
6. LinuxGetSsl



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.

To get an IP address

1. Parse the HTTP/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).

```
#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 either:

1. `bind(2)`, used by a server to specify where it will listen, or
2. `connect(2)`, used by a client to connect to a server.

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

```
#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.

```
#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`.

```
#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.

A `sockaddr *` is actually a generic pointer caste from one of several possible address structures, depending on the type of a connection. For an internet connection, you'll actually use a `sockaddr_in` (an internet `sockaddr`).

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

```
// 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 );
```

```
#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, nonblocking reads/writes and other options.

The only difference between `send()` and `write()` or between `recv()` and `read()` is the presence of 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 not mine. And pick a different name for your user-agent.

```
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 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 );
```

Here's the entire main() for LinuxGetUrl, minus only all the code.

```
#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.

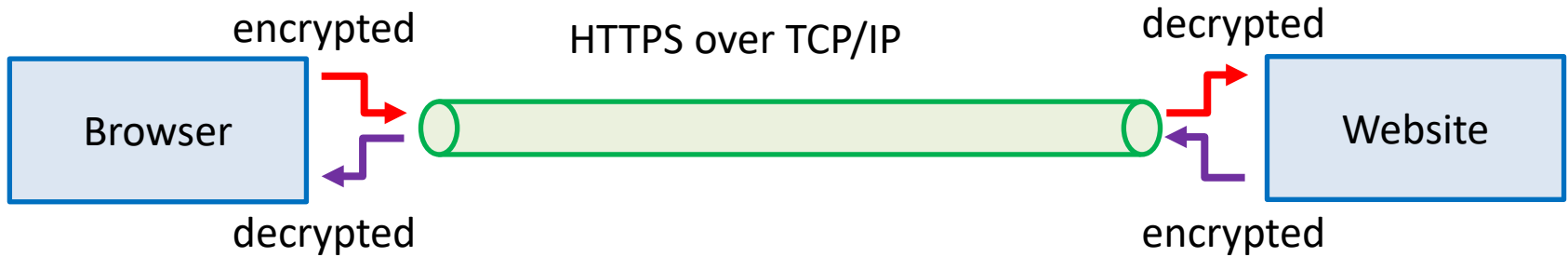
    // Send a GET message.

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

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

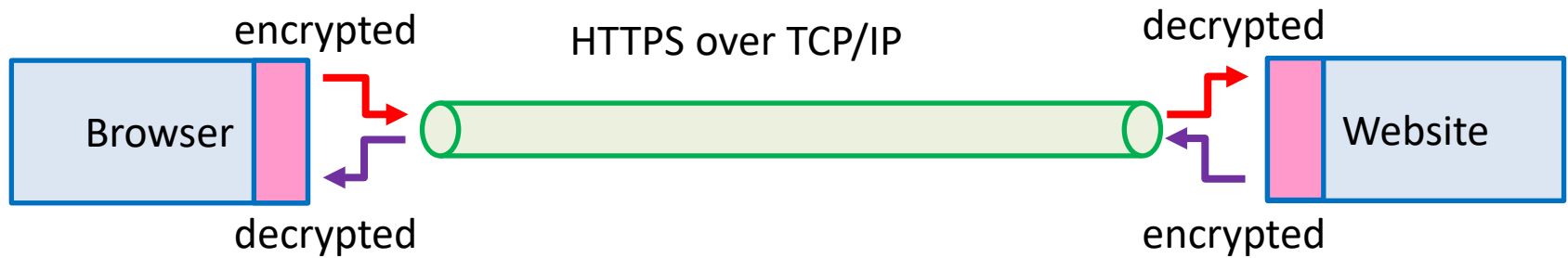
agenda

1. details
2. Host: parameter
3. Review of reading an HTTP webpage
4. Read an HTTPS webpage
5. TLS, SSL and the OpenSSL library
6. LinuxGetSsl



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.

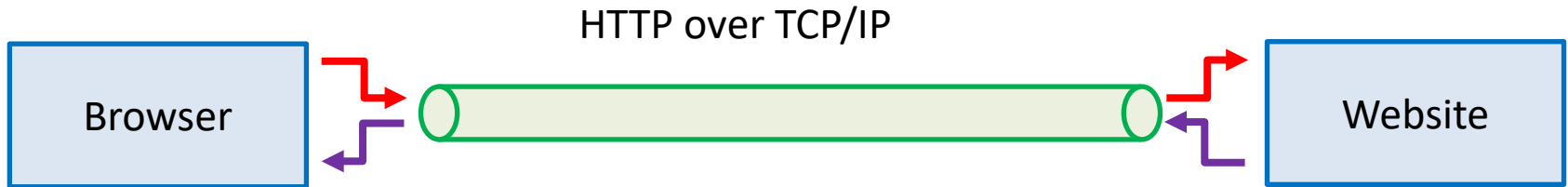
Done using a Secure Socket Layer (SSL) wrapper around a regular socket.



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.

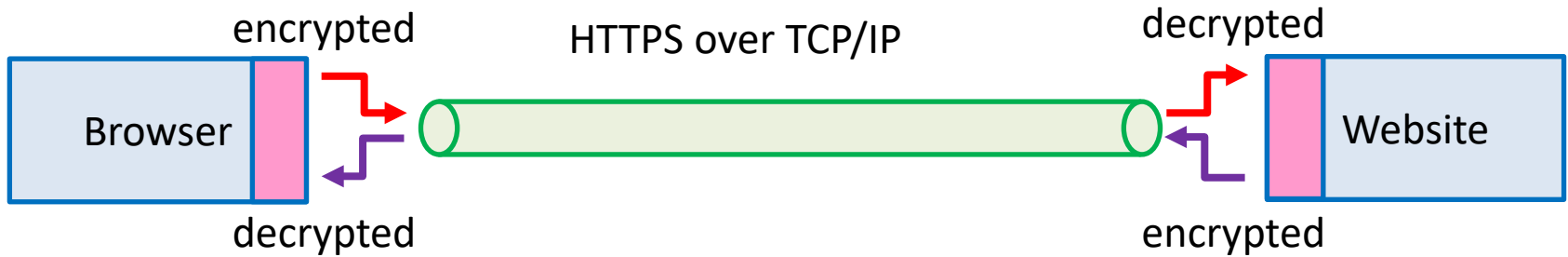
Done using a Secure Socket Layer (SSL) wrapper around a regular socket.

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. Send a GET message to request the page.
5. Read what comes back.



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.

secret communications

Traditionally, two parties would have to agree on a method and key for secret communications.

A book owned by both parties might be used with messages encrypted as references to page, line and word numbers, *PPPLLWW*.

Mechanical methods like the German Enigma relied on secret hardware and a key.

Problems:

1. You need a way of communicating securely how you'll do it before you can do it.
2. Secrecy depends on the secrecy of both the key and the method.

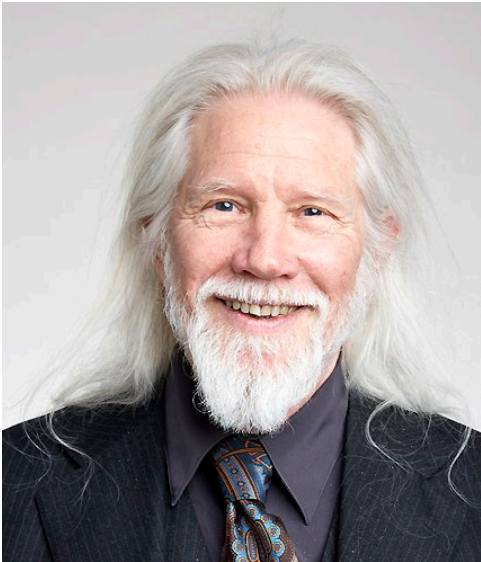


Image source: https://en.wikipedia.org/wiki/Enigma_machine

insights

1. The security of the system should only depend on secrecy of the key, not the secrecy of the method.
2. It should be possible for anyone to see how messages are encrypted, given the key, but without the key, knowing how it's done isn't helpful in breaking the message.

Diffie-Hellman key exchange, 1976



Whitfield Diffie



Martin Hellman



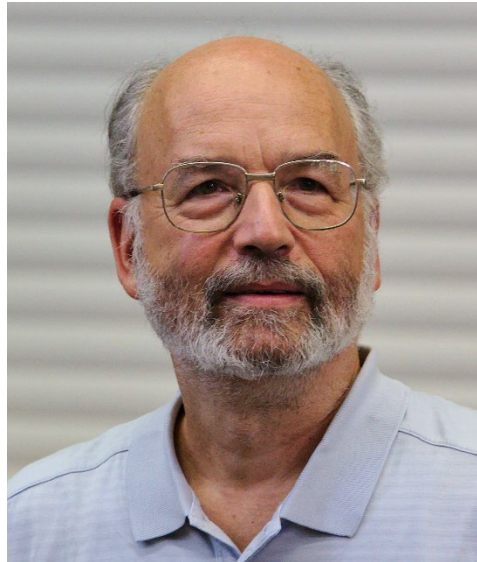
Ralph Merkle

Image sources: https://en.wikipedia.org/wiki/Whitfield_Diffie
https://en.wikipedia.org/wiki/Martin_Hellman
https://en.wikipedia.org/wiki/Ralph_Merkle

RSA public key encryption, 1978



Ron Rivest



Adi Shamir



Leonard Adleman

Image sources: https://en.wikipedia.org/wiki/Ron_Rivest
https://en.wikipedia.org/wiki/Adi_Shamir
https://en.wikipedia.org/wiki/Leonard_Adleman

public key encryption

Uses pairs of private and public keys that are related by a mathematical formula that's hard to reverse, e.g., factoring of very large numbers.

To get started:

1. I create (or pick) a private key which I keep secret,
2. I use that to create a public key, which I can share with the world.

If I want to send you an encrypted message:

1. I encrypt using my private key and your public key.
2. You decrypt using my public key and your private key.

Does not require a secure channel for initial exchange of secret keys.

encryption methods

Symmetric

Uses one key

Key must already be shared

Much faster

Short 128 to 256 bit keys

Asymmetric

Uses two keys, public and private

Does not require previous sharing

Slow

Very long 2048+ bit keys

SSL / TLS Handshake



public key encryption

RSA and Elliptic Curve Diffie-Hellman (ECDH/ECDHE) are the most commonly used PKE algorithms.

Modern TLS (1.3) heavily favors *ephemeral key exchanges* like ECDHE to provide forward secrecy.

An ephemeral key is a unique, temporary session key.

Forward secrecy (FS) is a security feature ensuring that a compromise of long-term server private keys does not compromise past session keys.

Using ephemeral key exchanges ensures that even if an attacker records encrypted traffic and later steals the server's private key, they cannot decrypt previously recorded messages.

once the handshake is complete

Once the TLS handshake establishes a shared secret, the subsequent data communication is encrypted using faster symmetric encryption.

AES-GCM (Advanced Encryption Standard with Galois/Counter Mode) is the most common modern standard.

Other commonly used algorithms include ChaCha20-Poly1305 and, in older configurations (TLS 1.2), AES-CBC.

agenda

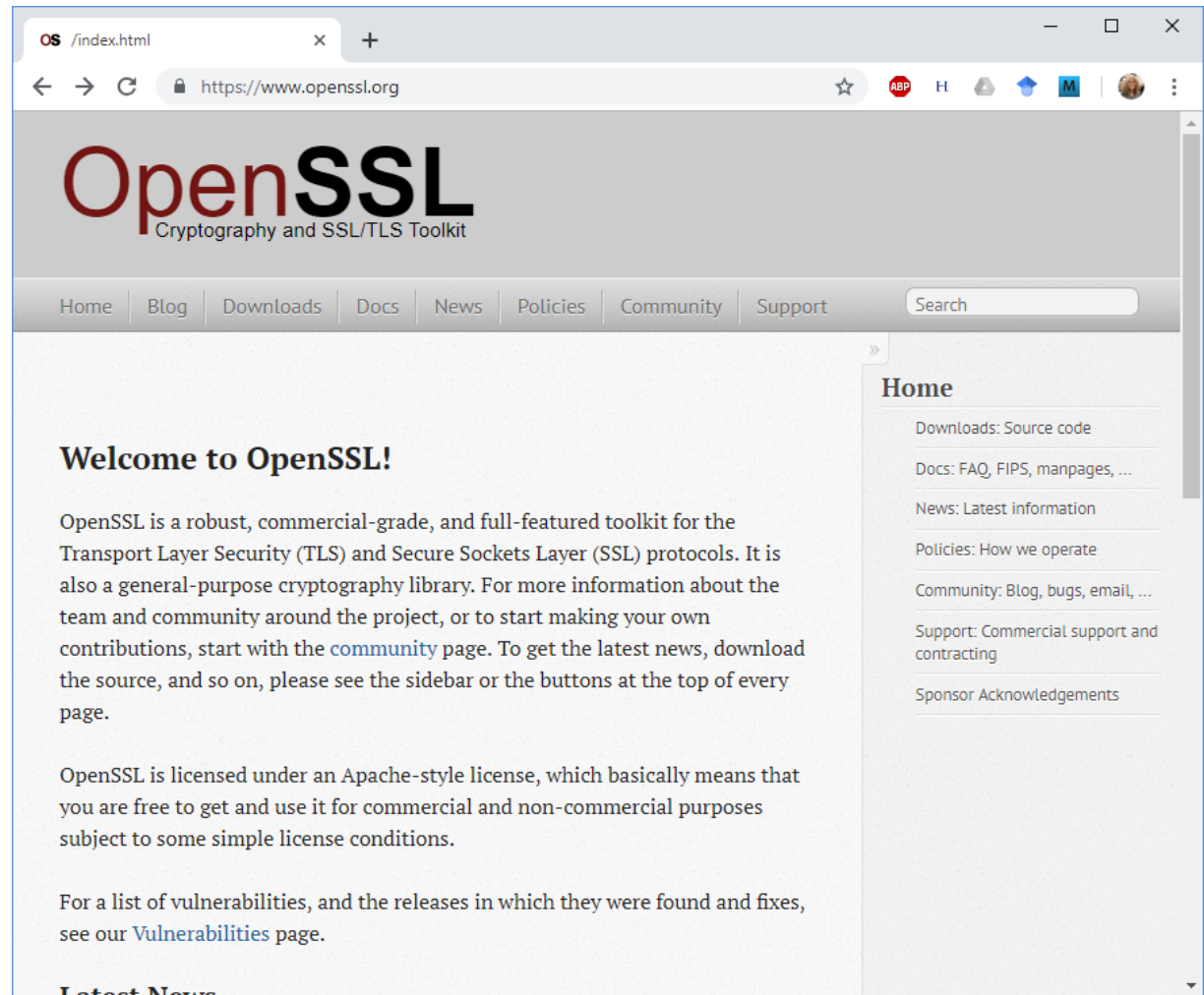
1. details
2. Host: parameter
3. Review of reading an HTTP webpage
4. Read an HTTPS webpage
5. TLS, SSL and the OpenSSL library
6. LinuxGetSsl

OpenSSL

The SSL/TLS handshake is too complex and rigorous to write on our own.

OpenSSL is a library you get to use.

It works on both Windows and Linux.



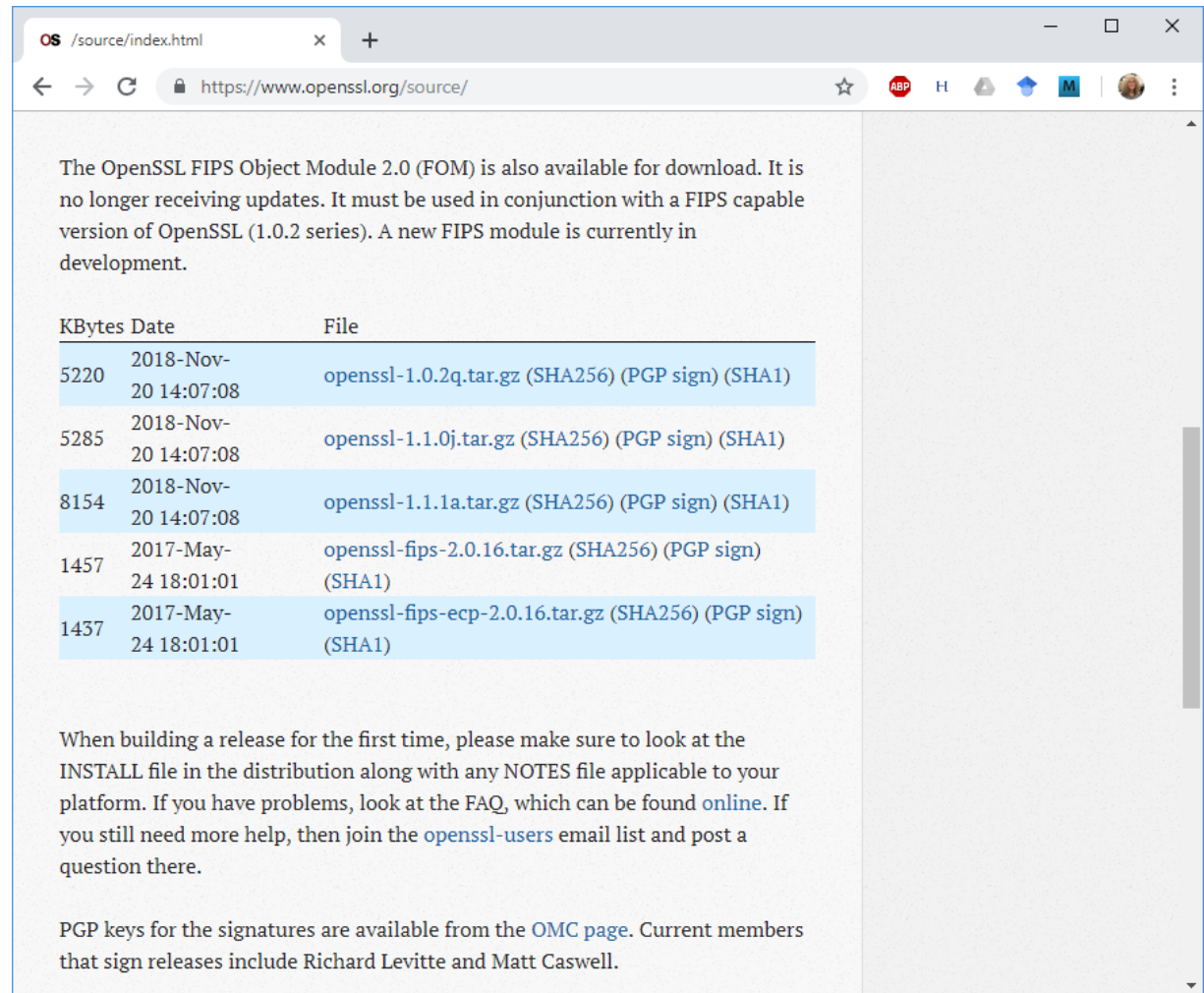
OpenSSL

But the downloads from OpenSSL are source only.

You have to build and install it yourself.

And it will not build in directory path with spaces, e.g., your Google drive path.

Alternative is to use `apt-get install`.

A screenshot of a web browser displaying the OpenSSL source page. The browser's address bar shows the URL 'https://www.openssl.org/source/'. The page content includes a paragraph about the OpenSSL FIPS Object Module 2.0 (FOM) and a table of available source files. The table has three columns: 'KBytes', 'Date', and 'File'. It lists five files, including 'openssl-1.0.2q.tar.gz', 'openssl-1.1.0j.tar.gz', 'openssl-1.1.1a.tar.gz', 'openssl-fips-2.0.16.tar.gz', and 'openssl-fips-ecp-2.0.16.tar.gz'. Below the table, there is a paragraph about building a release and a link to the FAQ. At the bottom, there is a paragraph about PGP keys and a link to the OMC page.

The OpenSSL FIPS Object Module 2.0 (FOM) is also available for download. It is no longer receiving updates. It must be used in conjunction with a FIPS capable version of OpenSSL (1.0.2 series). A new FIPS module is currently in development.

KBytes	Date	File
5220	2018-Nov-20 14:07:08	openssl-1.0.2q.tar.gz (SHA256) (PGP sign) (SHA1)
5285	2018-Nov-20 14:07:08	openssl-1.1.0j.tar.gz (SHA256) (PGP sign) (SHA1)
8154	2018-Nov-20 14:07:08	openssl-1.1.1a.tar.gz (SHA256) (PGP sign) (SHA1)
1457	2017-May-24 18:01:01	openssl-fips-2.0.16.tar.gz (SHA256) (PGP sign) (SHA1)
1457	2017-May-24 18:01:01	openssl-fips-ecp-2.0.16.tar.gz (SHA256) (PGP sign) (SHA1)

When building a release for the first time, please make sure to look at the [INSTALL](#) file in the distribution along with any [NOTES](#) file applicable to your platform. If you have problems, look at the [FAQ](#), which can be found [online](#). If you still need more help, then join the [openssl-users](#) email list and post a question there.

PGP keys for the signatures are available from the [OMC page](#). Current members that sign releases include Richard Levitte and Matt Caswell.

apt-get install OpenSSL

1. Install the OpenSSL library. `sudo apt-get install libssl-dev`
2. Include the openssl header. `#include <openssl/ssl.h>`
3. Compile and link the SSL and crypto libraries. `g++ LinuxGetSsl.cpp -lssl -lcrypto -o LinuxGetSsl`

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

```
int SSL_library_init(void);
```

SSL_library_init() initializes the SSL library.

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

```
int SSL_library_init(void);  
SSL_CTX *SSL_CTX_new(const SSL_METHOD *method);  
SSL *SSL_new(SSL_CTX *ctx);
```

SSL_library_init() initializes the SSL library.

SSL_CTX_new() creates a new SSL_CTX context object as framework to establish TLS/SSL enabled connections.

SSL_new() creates a new SSL structure need to hold the data for a TLS/SSL connection.

Here's an example initializing the SSL layer.

```
// Build the SSL layer.
```

```
SSL_library_init( );
```

```
SSL_CTX *ctx = SSL_CTX_new( SSLv23_method( ) );
```

```
assert( ctx );
```

```
SSL *ssl = SSL_new( ctx );
```

```
assert( ssl );
```

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

```
int SSL_set_fd(SSL *ssl, int fd);
```

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.

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

```
int SSL_connect(SSL *ssl);
```

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

Here's an example initializing the SSL layer.

```
// Fill in the socket we'll be using.
```

```
SSL_set_fd( ssl, s );
```

```
// Fill in the host we're trying to connect to.
```

```
SSL_set_tlsext_host_name( ssl, url.Host );
```

```
// Establish an SSL connection.
```

```
int sslConnectResult = SSL_connect( ssl );
```

```
assert( sslConnectResult == 1 );
```

The requirement to fill in the host is new.

Apache servers got an update in 2025 that now requires that you fill in the host name in the SSL.

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

```
int SSL_write(SSL *ssl, const void *buf, int num);
```

```
int SSL_read(SSL *ssl, void *buf, int num);
```

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 an example reading and writing.

```
while ( ( bytes = SSL_read( ssl, buffer,  
    sizeof( buffer ) ) ) > 0 )  
    write( 1, buffer, bytes );
```

Shutdown and free up resources at the end.

```
SSL_shutdown( ssl );  
SSL_free( ssl );  
SSL_CTX_free( ctx );
```

agenda

1. details
2. Host: parameter
3. Review of reading an HTTP webpage
4. Read an HTTPS webpage
5. TLS, SSL and the OpenSSL library
6. **LinuxGetSsl**

Agenda

1. Course details.
2. Correction on the Host: parameter.
3. Review of reading an HTTP webpage.
4. Read an HTTPS webpage.
5. TLS, SSL and the OpenSSL library.
6. **LinuxGetSSL.**

Here's the entire main() for LinuxGetSsl, minus only all the code.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <openssl/ssl.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 socket we've connected.

    // Send a GET message.

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

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