EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Bonus lecture: TCP/IP DNS sockets and servers

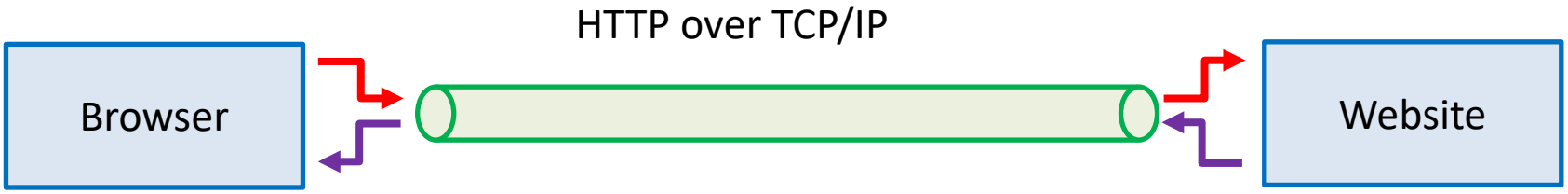
Nicole Hamilton
https://web.eecs.umich.edu/~nham/
nham@umich.edu

Agenda

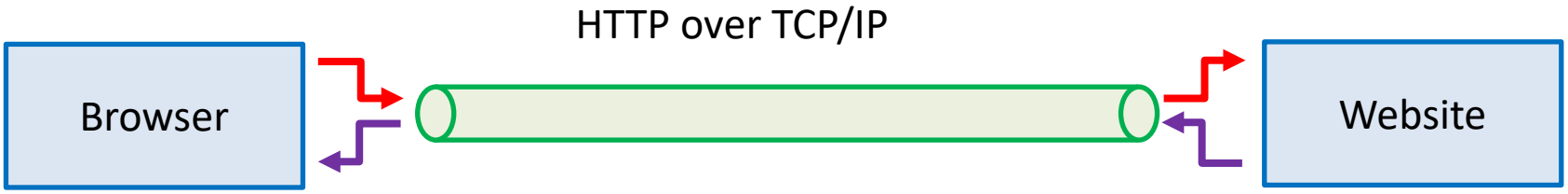
1. Read a webpage.
2. TCP/IP.
3. DNS.
4. Sockets.
5. LinuxGetUrl
6. Read an HTTPS webpage.
7. TLS, SSL and the OpenSSL library.
8. LinuxGetSSL.
9. TinyLinuxServer.
10. bind( ), listen( ) and accept( ).
11. The Talk( ) thread.
12. A plugin interface.

Agenda

1. Read a webpage.
2. TCP/IP.
3. DNS.
4. Sockets.
5. LinuxGetUrl
6. Read an HTTPS webpage.
7. TLS, SSL and the OpenSSL library.
8. LinuxGetSSL.
9. TinyLinuxServer.
10. bind( ), listen( ) and accept( ).
11. The Talk( ) thread.
12. A plugin interface.



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 I assign in my search engine class:

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. Read a webpage.
2. TCP/IP.
3. DNS.
4. Sockets.
5. LinuxGetUrl
6. Read an HTTPS webpage.
7. TLS, SSL and the OpenSSL library.
8. LinuxGetSSL.
9. TinyLinuxServer.
10. bind( ), listen( ) and accept( ).
11. The Talk( ) thread.
12. A plugin interface.

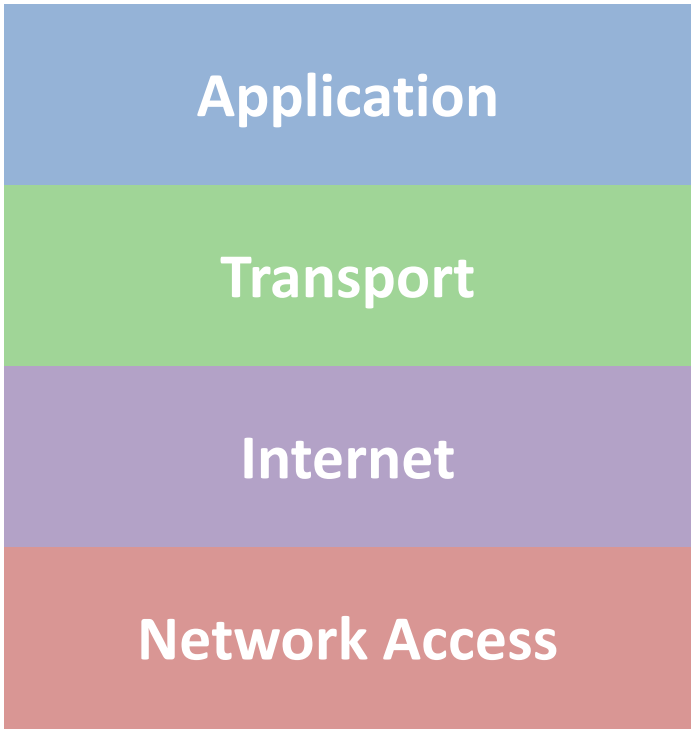
TCP/IP Model

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



Applications

DHCP	Dynamic Host Configuration Protocol. A DHCP server assigns an IP address to each device on a network.
DNS	Domain Name System. A decentralized naming system that allows domain names to be translated to IP addresses.
FTP	File Transfer Protocol.
HTTP	Hypertext Transfer Protocol for web browsers and web servers.
HTTPS	Hypertext Transfer Protocol Secure.
POP	Post Office Protocol for transferring email.
SMTP	Simple Mail Transfer Protocol.
SSH	Secure Shell for remote login.

Transport protocols

TCP	Transmission Control Protocol. Reliable, ordered, error-checked delivery of a byte stream.
UDP	User Datagram Protocol. Connectionless communication. Checksums for error detection. No guarantee of delivery, ordering or protection from duplicates.

IP addresses

IPv4	Internet Protocol version 4. Uses a 32-bit big-endian address space. Uses a dotted notation, each group of 8 bits, starting at the high end, written as a decimal number, e.g., umich.edu = `141.211.243.251`. Now facing *address exhaustion.*
IPv6	Internet Protocol version 6. Uses a 128-bit big-endian address. Written as 8 groups of 4 hex characters separated by colons. If a group is all zeros, it can be omitted, e.g., `2001:db8::8a2e:370:7334`.

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. Read a webpage.
2. TCP/IP.
3. DNS.
4. Sockets.
5. LinuxGetUrl
6. Read an HTTPS webpage.
7. TLS, SSL and the OpenSSL library.
8. LinuxGetSSL.
9. TinyLinuxServer.
10. bind( ), listen( ) and accept( ).
11. The Talk( ) thread.
12. A plugin interface.

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).

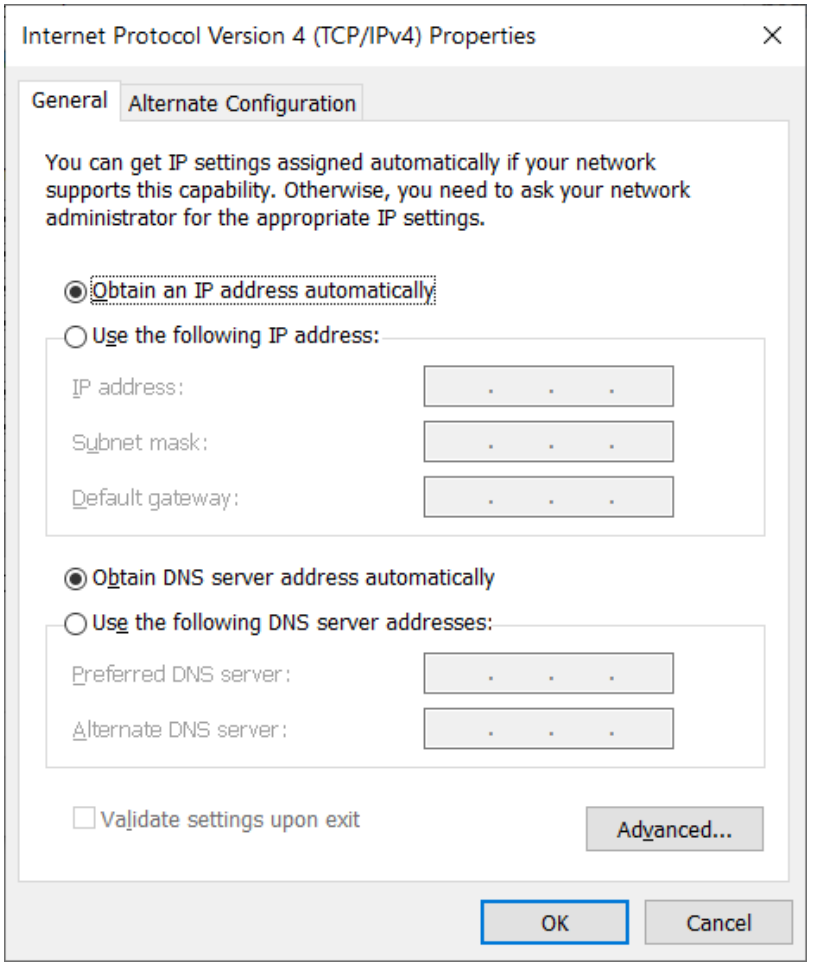
DNS Records

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.



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
```

```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).

```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).

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 assuming an IPv4 address.

```
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. Read a webpage.
2. TCP/IP.
3. DNS.
4. Sockets.
5. LinuxGetUrl
6. Read an HTTPS webpage.
7. TLS, SSL and the OpenSSL library.
8. LinuxGetSSL.
9. TinyLinuxServer.
10. bind( ), listen( ) and accept( ).
11. The Talk( ) thread.
12. A plugin interface.

```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

```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 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, noblocking 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 write().

Here's a sample GET message we might send.

```
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.

(In 398, I required students to put their contact info into the User-Agent field so complaints would go to them, not me.)

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

Host: parameter

Lots of servers host lots of websites at the same IP address and port number.

They distinguish which website you mean by the Host: parameter.

So, it’s not redundant.

My websites are on a GoDaddy machine with lots of other websites.

```
tcsh-2% ./LinuxGetSsl https://nicolehamilton.com
Service = https, Host = nicolehamilton.com, Port = , Path =
Host address length = 16 bytes
Family = 2, port = 443, address = 160.153.46.5
GET / HTTP/1.1
Host: nicolehamilton.com
User-Agent: LinuxGetSsl/2.0 nham@umich.edu (Linux)
Accept: */*
Accept-Encoding: identity
Connection: close


HTTP/1.1 200 OK
Date: Thu, 19 Sep 2019 17:02:10 GMT
Server: Apache
Upgrade: h2,h2c
Connection: Upgrade, close
Last-Modified: Thu, 11 Oct 2018 21:59:57 GMT
ETag: "c4206db-3f6f-577fb177483a1"
Accept-Ranges: bytes
Content-Length: 16239
:
```

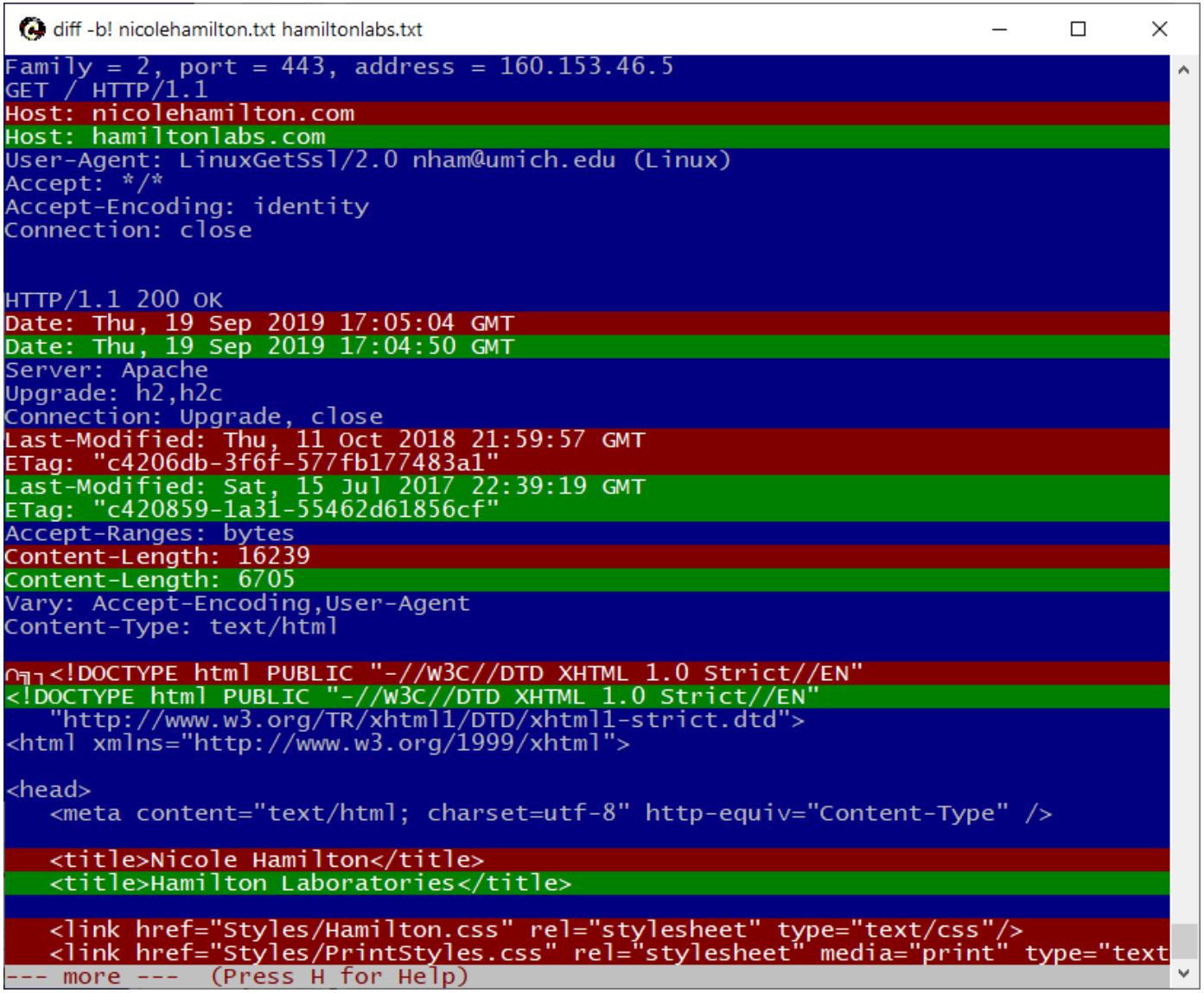
So, both nicolehamilton.com and hamiltonlabs.com are at 160.153.46.5:443.

```
tcsh-2% ./LinuxGetSsl https://nicolehamilton.com
Service = https, Host = nicolehamilton.com, Port = , Path =
Host address length = 16 bytes
Family = 2, port = 443, address = 160.153.46.5
GET / HTTP/1.1
Host: nicolehamilton.com
User-Agent: LinuxGetSsl/2.0 nham@umich.edu (Linux)
Accept: */*
Accept-Encoding: identity
Connection: close


HTTP/1.1 200 OK
Date: Thu, 19 Sep 2019 17:02:10 GMT
Server: Apache
Upgrade: h2,h2c
Connection: Upgrade, close
Last-Modified: Thu, 11 Oct 2018 21:59:57 GMT
ETag: "c4206db-3f6f-577fb177483a1"
Accept-Ranges: bytes
Content-Length: 16239
:
```

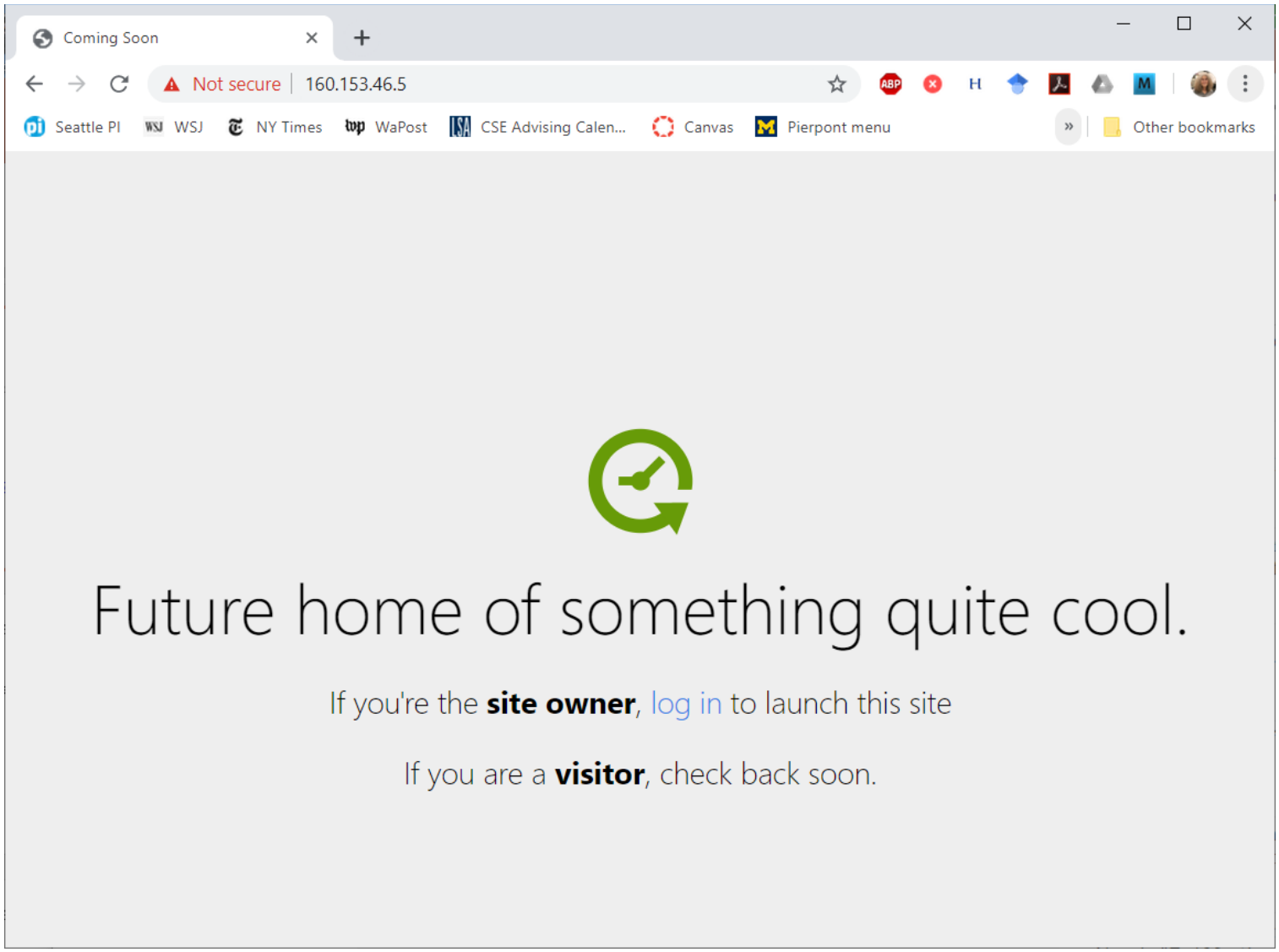
Both nicolehamilton.com and hamiltonlabs.com are at 160.153.46.5:443.

```
tcsh-3% ./LinuxGetSsl https://hamiltonlabs.com
Service = https, Host = hamiltonlabs.com, Port = , Path =
Host address length = 16 bytes
Family = 2, port = 443, address = 160.153.46.5
GET / HTTP/1.1
Host: hamiltonlabs.com
User-Agent: LinuxGetSsl/2.0 nham@umich.edu (Linux)
Accept: */*
Accept-Encoding: identity
Connection: close


HTTP/1.1 200 OK
Date: Thu, 19 Sep 2019 17:03:31 GMT
Server: Apache
Upgrade: h2,h2c
Connection: Upgrade, close
Last-Modified: Sat, 15 Jul 2017 22:39:19 GMT
ETag: "c420859-1a31-55462d61856cf"
Accept-Ranges: bytes
Content-Length: 6705
:
```

The server response depends on which Host: was specified.



If you specify Host: 160.153.46.5, you get GoDaddy's login page for that server.



Here's an example sending the Get message.

```
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 );
```

Agenda

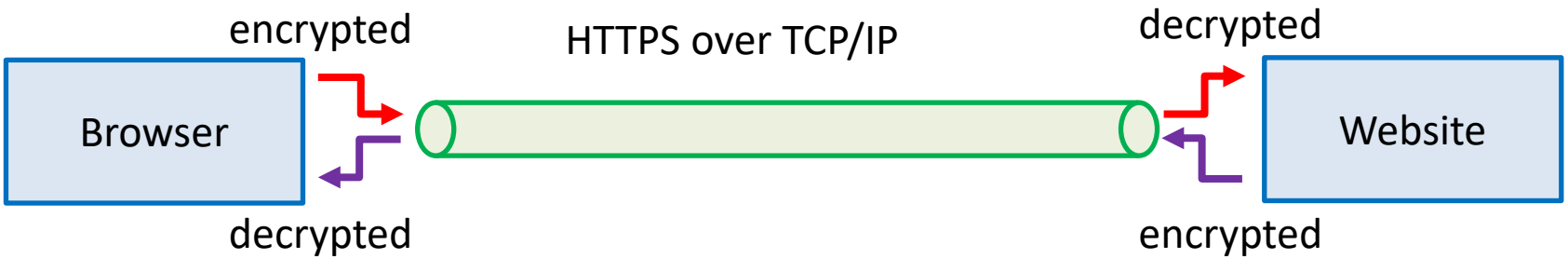
1. Read a webpage.
2. TCP/IP.
3. DNS.
4. Sockets.
5. LinuxGetUrl
6. Read an HTTPS webpage.
7. TLS, SSL and the OpenSSL library.
8. LinuxGetSSL.
9. TinyLinuxServer.
10. bind( ), listen( ) and accept( ).
11. The Talk( ) thread.
12. A plugin interface.

Here's the entire main( ), 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.
   }
```

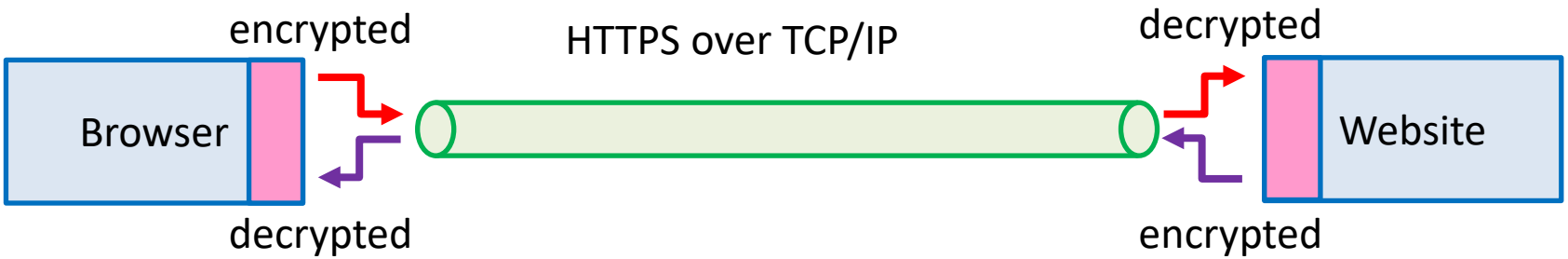
Agenda

1. Read a webpage.
2. TCP/IP.
3. DNS.
4. Sockets.
5. LinuxGetUrl
6. Read an HTTPS webpage.
7. TLS, SSL and the OpenSSL library.
8. LinuxGetSSL.
9. TinyLinuxServer.
10. bind( ), listen( ) and accept( ).
11. The Talk( ) thread.
12. A plugin interface.



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.

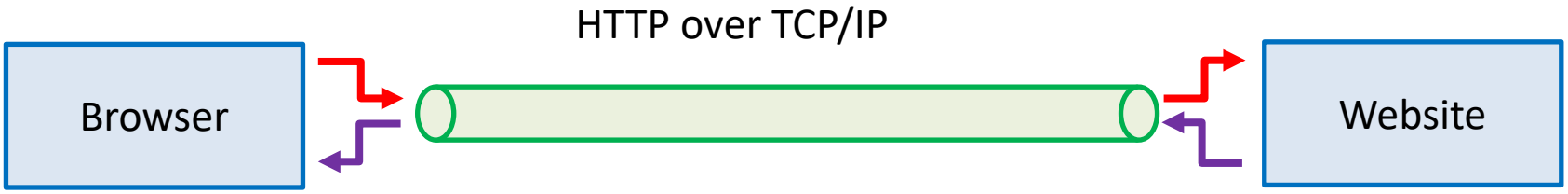
Done using a Secure Socket Layer (SSL) wrapper around a regular socket.



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.

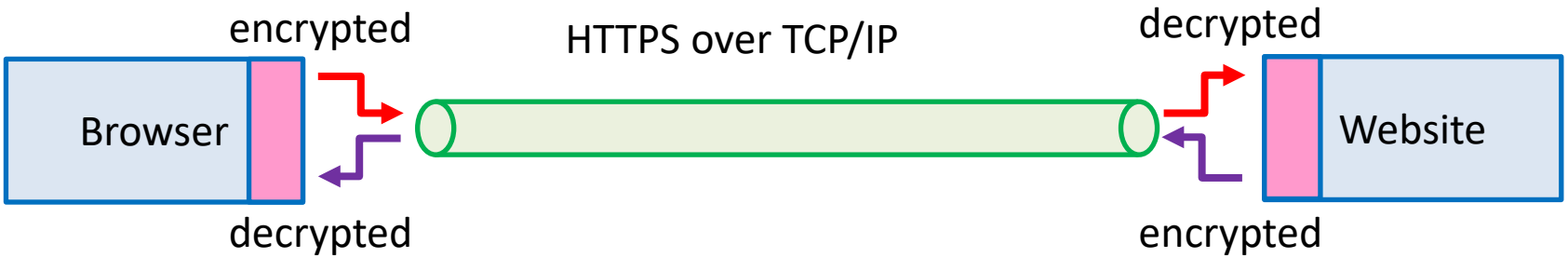
Done using a Secure Socket Layer (SSL) wrapper around a regular socket.

Here, we'll use the OpenSSL library.



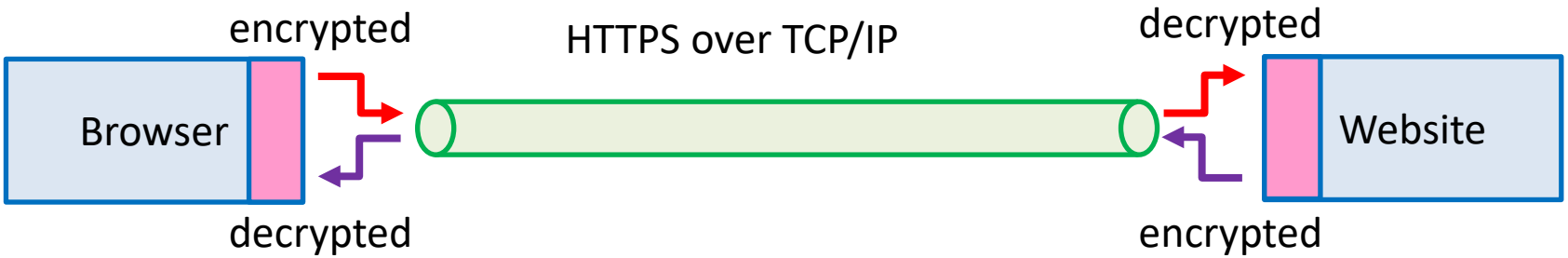
To read a page from a website:

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.



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.

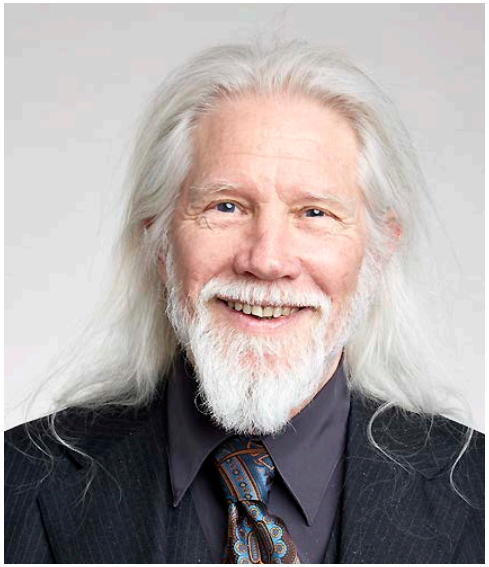
2. Secrecy depends on the secrecy of both the key and the method.

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

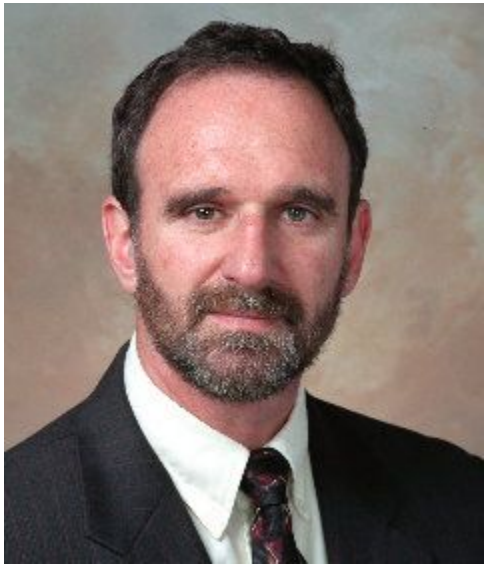
Insights

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

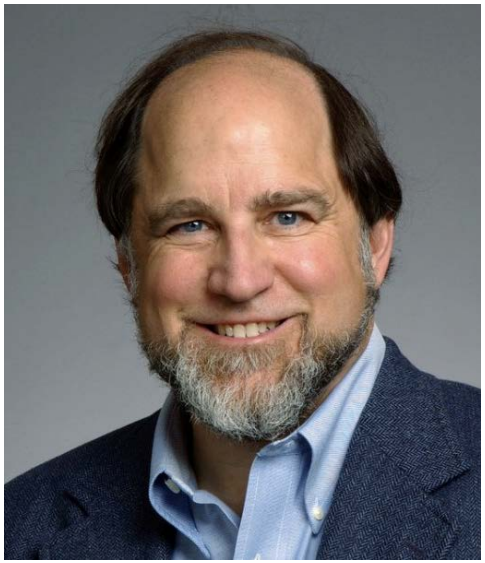


Martin Hellman

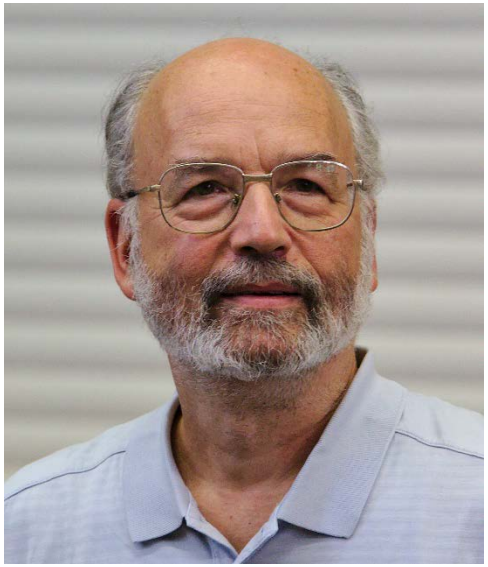
Ralph Merkle

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

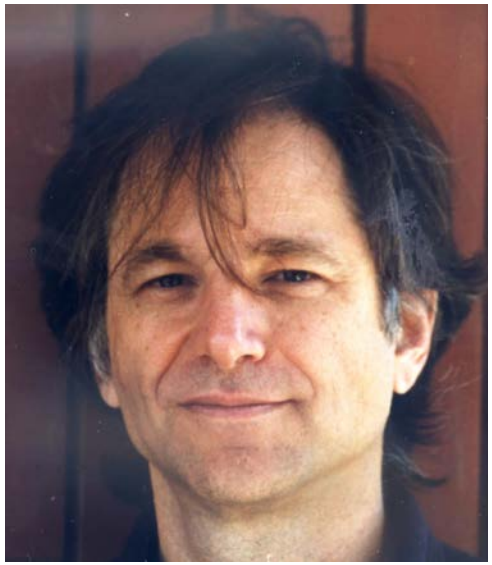
RSA public key encryption, 1978



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, factoring of very large numbers.

To get started:

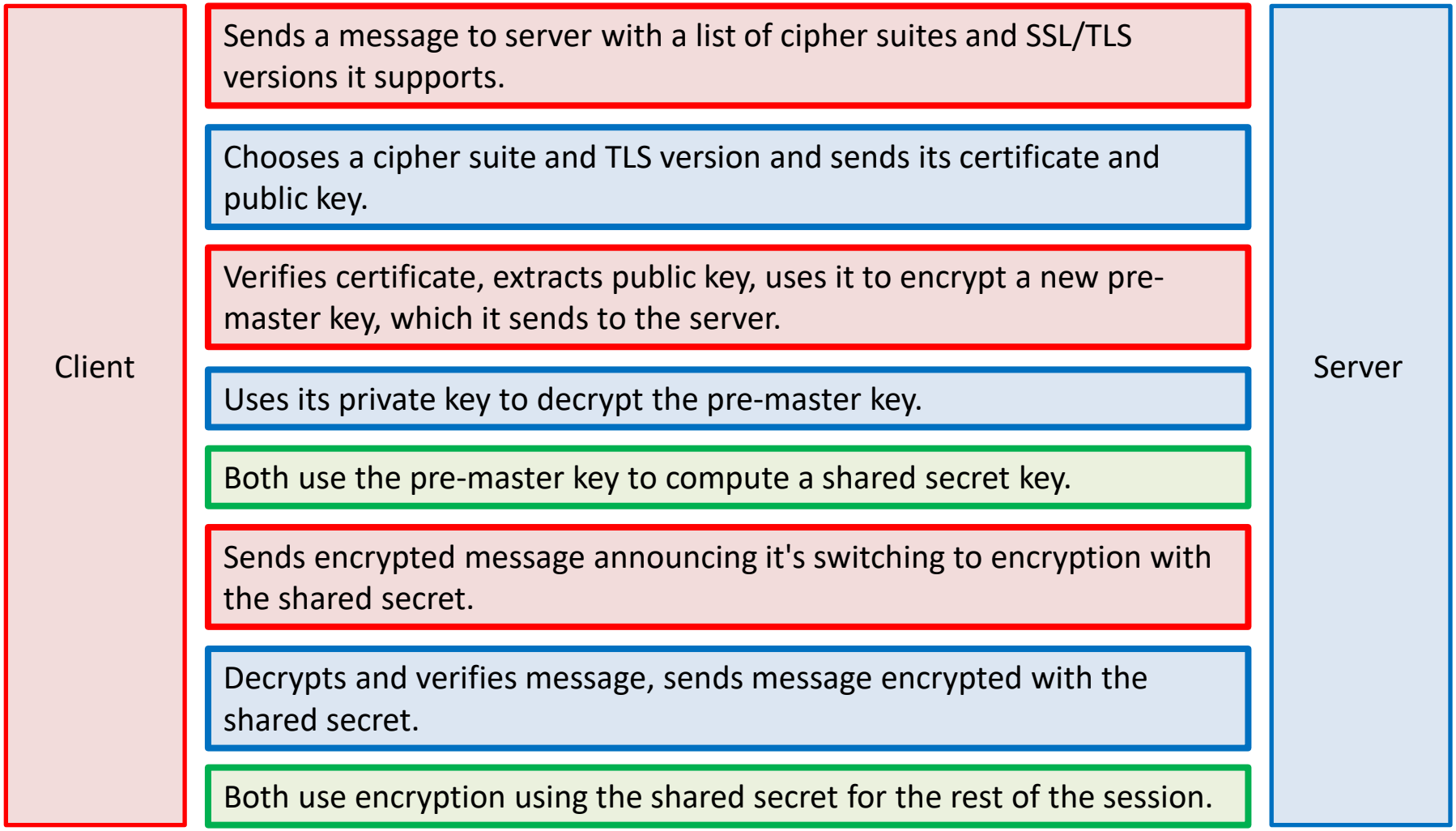
1. I create (or pick) a private key which I keep secret,
2. then use that to create a public key, which I can share with the world.

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.

SSL / TLS Handshake



Agenda

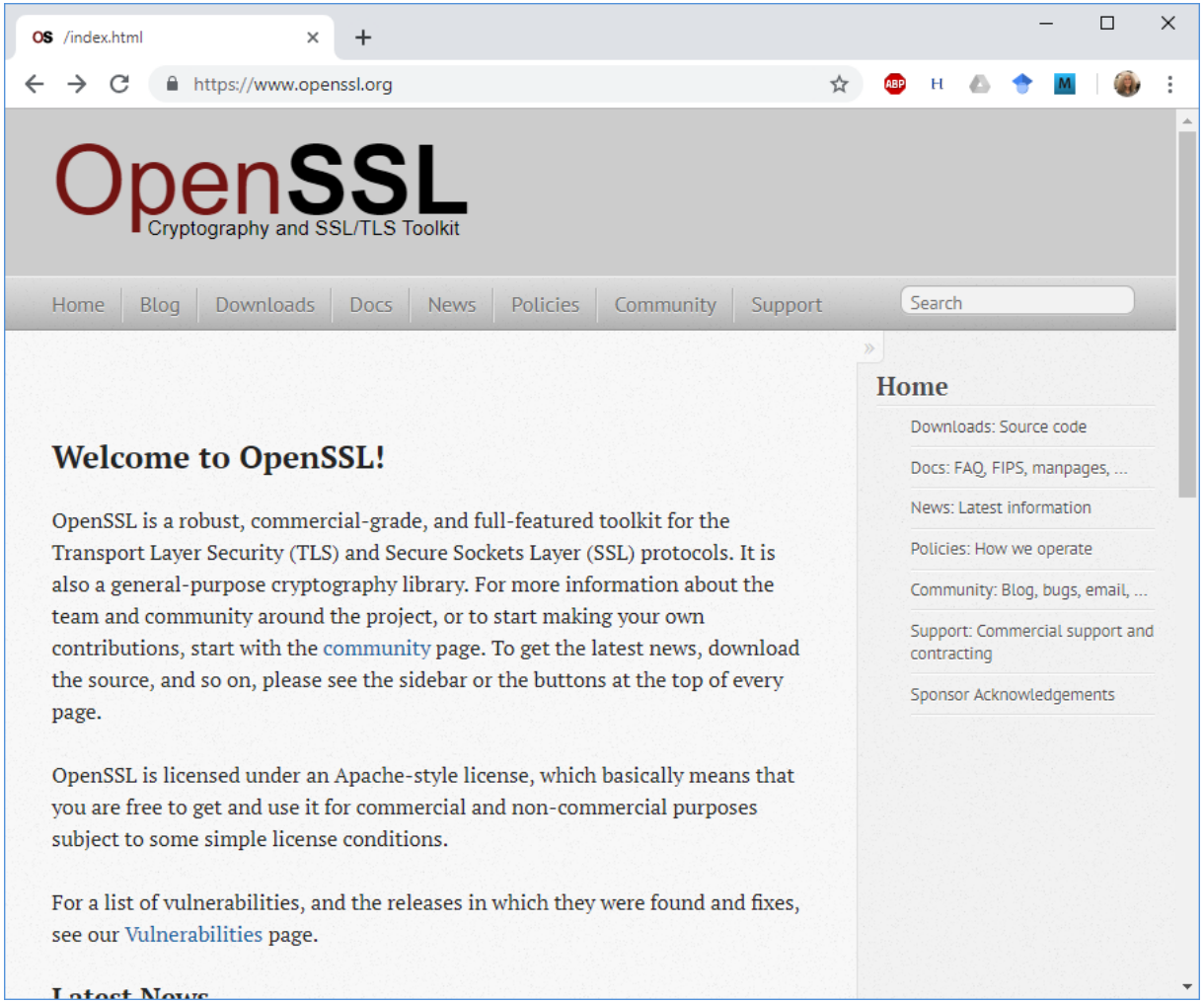
1. Read a webpage.
2. TCP/IP.
3. DNS.
4. Sockets.
5. LinuxGetUrl
6. Read an HTTPS webpage.
7. TLS, SSL and the OpenSSL library.
8. LinuxGetSSL.
9. TinyLinuxServer.
10. bind( ), listen( ) and accept( ).
11. The Talk( ) thread.
12. A plugin interface.

OpenSSL

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

OpenSSL is a popular library.

It works on both Windows and Linux.



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
```

```c
#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);
int SSL_CTX_free(SSL_CTX *ctx);
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_CTX_free() frees the SSL_CTX object.
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 );
```

```c
#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.

```c
#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 );

// Establish an SSL connection.

int sslConnectResult = SSL_connect( ssl );
assert( sslConnectResult == 1 );
```

```c
#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. Read a webpage.
2. TCP/IP.
3. DNS.
4. Sockets.
5. LinuxGetUrl
6. Read an HTTPS webpage.
7. TLS, SSL and the OpenSSL library.
8. LinuxGetSSL.
9. TinyLinuxServer.
10. bind( ), listen( ) and accept( ).
11. The Talk( ) thread.
12. A plugin interface.

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.
   }
```

Agenda

1. Read a webpage.
2. TCP/IP.
3. DNS.
4. Sockets.
5. LinuxGetUrl
6. Read an HTTPS webpage.
7. TLS, SSL and the OpenSSL library.
8. LinuxGetSSL.
9. TinyLinuxServer.
10. bind( ), listen( ) and accept( ).
11. The Talk( ) thread.
12. A plugin interface.

A simple web server

Two parts:

1. The HTTP server.
2. A plugin that can exchange JSON with a webpage.

LinuxTinyServer

A very minimal web server for Linux.

1. It begins listing on a socket for connection requests from browser.
2. Each time it gets a request, it creates a thread with a new socket to talk to the client.
3. If it's a "magic path", it can call a plugin module.
4. Otherwise, it handles GET requests by serving up the specified file if it exists.

LinuxTinyServer takes a port number and a root directory for a website.

```
tcsh-5% 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-6% ./LinuxTinyServer
Usage:  ./LinuxTinyServer port rootdirectory
tcsh-7%
```

LinuxTinyServer opens a socket and begins listening for connections.

```
tcsh-6% ./LinuxTinyServer
Usage:  ./LinuxTinyServer port rootdirectory
tcsh-7% ls website
Images  Styles  index.htm
tcsh-8% ./LinuxTinyServer 5000 website
Listening on 0.0.0.0:5000
```

LinuxTinyServer responds with the requested page. If you want index.htm, you must ask for it. LTS will not look for it if you give only a directory path.

```
tcsh-8% ./LinuxTinyServer 5000 website
Listening on 0.0.0.0:5000

Connection accepted from 127.0.0.1:54690

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
```

Agenda

1. Read a webpage.
2. TCP/IP.
3. DNS.
4. Sockets.
5. LinuxGetUrl
6. Read an HTTPS webpage.
7. TLS, SSL and the OpenSSL library.
8. LinuxGetSSL.
9. TinyLinuxServer.
10. bind( ), listen( ) and accept( ).
11. The Talk( ) thread.
12. A plugin interface.

bind( ), listen( ) and accept( )

The basic steps to a web server:

1. Creates two socket variables, one for listening, the other when a new client connects.
2. Build a sockaddr_in structure specifying internet protocol, port number, any IP address, TCP stream.
3. Binds the socket to that address.
4. Enters a loop where it begins listening.
5. Each time it gets a connection request it spawns a thread to talk to the client.

bind( )

bind( ) is used to connect a socket to a particular address, protocol and port where it can listen.

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

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

bind( )

bind( ) is used to connect a socket to a particular address, protocol and port where it can listen.

All of that is specified in an addrinfo structure.

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

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

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;
};
```

getsockname( )

If you specify port 0, the system will assign one.

Here's how you can find out what you were assigned.

```
struct sockaddr_in listenPort;
socklen_t listenPortLength = sizeof( listenPort );
memset( &listenPort, 0, sizeof( listenPort ) );
int getsocknameResult = getsockname( listenSocket,
      ( sockaddr * )&listenPort,    &listenPortLength );
assert( getsocknameResult != -1 );
port = ntohs( listenPort.sin_port );
```

listen( )

listen( ) marks the socket as one to be used for accepting incoming connection requests.

The backlog is the maximum queue length of pending connections.

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

int listen(int sockfd, int backlog);
```

listen( )

listen( ) marks the socket as one to be used for accepting incoming connection requests.

The backlog is the maximum queue length of pending connections.

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

int listen(int sockfd, int backlog);
```

```
SOMAXCONN is a system-configured default
maximum socket queue length.

(Under WSL Ubuntu, it's defined as 128 in
/usr/include/x86_64-linux-gnu/bits/socket.h.)
```

listen( )

Any client anywhere on the web that has your IP and port address can try to connect to you.

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

int listen(int sockfd, int backlog);
```

SOMAXCONN is a system-configured default maximum socket queue length.

(Under WSL Ubuntu, it's defined as 128 in /usr/include/x86_64-linux-gnu/bits/socket.h.)

accept( )

accept( ) waits until a client tries to do a connect( ) and then returns socket file descriptor that's created.

The sockaddr that's returns tells you the client's IP address.

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

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

```
int main( int argc, char **argv )
   {
   // Check usage and arguments.

   // Create two sockets, one for listening for new
   // connection requests, the other for talking to each
   // new client.

   // Create socket address structures to go with each
   // socket, filling in details of where we'll listen.

   // Create the listenSocket, specifying that we'll r/w
   // it as a stream of bytes using TCP/IP.

   // Bind the listen socket to the IP address and protocol
   // where we'd like to listen for connections.

   // Begin listening for clients to connect to us.

   // Accept each new connection and create a thread to talk with
   // the client over the new talk socket that's created by Linux
   // when we accept the connection.

   // Close the listen socket.
   }
```

Passing the talk socket to the child

When creating a child thread, you get to pass a void *, usually a pointer to an object with whatever information the child needs.

Since the server expects to get lots of connection requests, it can't pass a pointer to a local or global variable that will quickly be overwritten.

Solution is to pass a pointer to an object on the heap and let the child delete it.

```
while ( ( talkAddressLength = sizeof( talkAddress ),
      talkSocket = accept( ... ) ) && talkSocket != -1 )
   {
   pthread_t child;
   pthread_create( &child, nullptr, Talk,
         new int( talkSocket ) );
   pthread_detach( child );
   }
```

```c
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
        void *(*start_routine) (void *), void *arg);

int pthread_join(pthread_t thread, void **retval);

int pthread_detach(pthread_t thread);
```

Compile and link with -pthread.

The pthread_create() function starts a new thread in the calling process. The new thread starts execution by invoking start_routine(); arg is passed as the sole argument of start_routine().

The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately.

pthread_detach() causes the thread's resources to be released automatically when the thread terminates.

Using boost

Here's how you do it with boost threads, which let you pass objects, not just a void pointer.

Install boost using:
sudo apt-get install libboost-all-dev

```
#include <boost/thread/thread.hpp>

while ( ( talkAddressLength = sizeof( talkAddress ),
      talkSocket = accept( ... ) ) && talkSocket != -1 )
   {
   boost::thread talkThread( Talk, talkSocket );
   talkThread.detach( );
   }
```

Agenda

1. Read a webpage.
2. TCP/IP.
3. DNS.
4. Sockets.
5. LinuxGetUrl
6. Read an HTTPS webpage.
7. TLS, SSL and the OpenSSL library.
8. LinuxGetSSL.
9. TinyLinuxServer.
10. bind( ), listen( ) and accept( ).
11. The Talk( ) thread.
12. A plugin interface.

The talk thread

The Talk thread looks for a GET message and replies with the requested file.

But it also has a plugin interface that allows a server application to intercept “magic paths”.

```
void *Talk( void *talkSocket )
   {
   // Cast from void * to int * to recover the talk
   // socket id then delete the copy passed on the heap.

   // Allocate a buffer for reading the incoming
   // request  and for reading the requested file.

   // Do a recv( ) to get the request.

   // Parse the request to find the action and path
   // being requested.

   // Watch for a plugin that intercepts this path.

   // If it's a GET and the path is found in the website
   // directory, return it with an HTTP/1.1 200 OK
   // message, otherwise with a 403 or 404.

   // Close the talk socket.
   }
```

The talk thread

It can't just paste the requested path onto the end of the website directory path.

Must watch for ".." segments and forbid access outside the website.

```
void *Talk( void *talkSocket )
   {
   // Cast from void * to int * to recover the talk
   // socket id then delete the copy passed on the heap.

   // Allocate a buffer for reading the incoming
   // request  and for reading the requested file.

   // Do a recv( ) to get the request.

   // Parse the request to find the action and path
   // being requested.

   // Watch for a plugin that intercepts this path.

   // If it's a GET and the path is found in the website
   // directory, return it with an HTTP/1.1 200 OK
   // message, otherwise with a 403 or 404.

   // Close the talk socket.
   }
```

The boost version

In the boost version, arguments are passed directly, not as a void pointer.

```
void Talk( int talkSocket )
   {
   // Allocate a buffer for reading the incoming
   // request  and for reading the requested file.

   // Do a recv( ) to get the request.

   // Parse the request to find the action and path
   // being requested.

   // Watch for a plugin that intercepts this path.

   // If it's a GET and the path is found in the website
   // directory, return it with an HTTP/1.1 200 OK
   // message, otherwise with a 403 or 404.

   // Close the talk socket.
   }
```

Agenda

1. Read a webpage.
2. TCP/IP.
3. DNS.
4. Sockets.
5. LinuxGetUrl
6. Read an HTTPS webpage.
7. TLS, SSL and the OpenSSL library.
8. LinuxGetSSL.
9. TinyLinuxServer.
10. bind( ), listen( ) and accept( ).
11. The Talk( ) thread.
12. A plugin interface.

The plugin interface

The Talk thread looks for a GET message and replies with the requested file.

But it also has a plugin interface that allows a server application to intercept “magic paths”.

```
class PluginObject
   {
   public:
      // MagicPath returns true if this is a path
      // the plugin intercepts.

      virtual bool MagicPath( string path ) = 0;

      // The request passed to ProcessRequest is
      // the raw contents of the HTTP request as
      // read from the talk socket.

      // Whatever is returned is  written unchanged
      // to the socket (and to the client) with a
      // proper HTTP header.

      string ProcessRequest( string request ) = 0;

      virtual ~PluginObject( )
         {
         }
};
```

The plugin interface

The plugin registers itself by setting a global pointer.

```
// The constructor for any plugin should set
// Plugin = this so that LinuxTinyServer knows
// it exists and can call it.

extern PluginObject *Plugin;
```

The plugin interface

The plugin registers itself by setting a global pointer.

The initial value is null.

```
// The constructor for any plugin should set
// Plugin = this so that LinuxTinyServer knows
// it exists and can call it.

#include "Plugin.h"
PluginObject *Plugin = nullptr;
```

The plugin interface

Example: The new EECS 280 P4 Web project.

The plugin constructor registers itself by setting the global pointer.

https://eecs280staff.github.io/p4-web/

```
class P4_Web : public PluginObject
   {
   public:
      bool MagicPath( const string path )
         {
         // Return true if this is a path that
         // this plugin intercepts.
         }
      string ProcessRequest( const string request )
         {
         // Read the request and return a string
         // with the proper HTTP header and content.
         }
      P4_Web( )
         {
         // Register this plugin.
         Plugin = this;
         }
      ~P4_Web( )
         {
         }
};
```