

EECS 498 System Design of a Search Engine

Winter 2026

Lecture 12: LinuxTinyServer

Nicole Hamilton

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

agenda

1. details
2. HTTP redirects
3. LinuxTinyServer
4. opening ports to WSL2
5. `bind( )`, `listen( )` and `accept( )`
6. the `Talk( )` thread
7. a plugin interface
8. forms

agenda

1. details
2. HTTP redirects
3. LinuxTinyServer
4. opening ports to WSL2
5. `bind( )`, `listen( )` and `accept( )`
6. the `Talk( )` thread
7. a plugin interface
8. forms

details

1. none

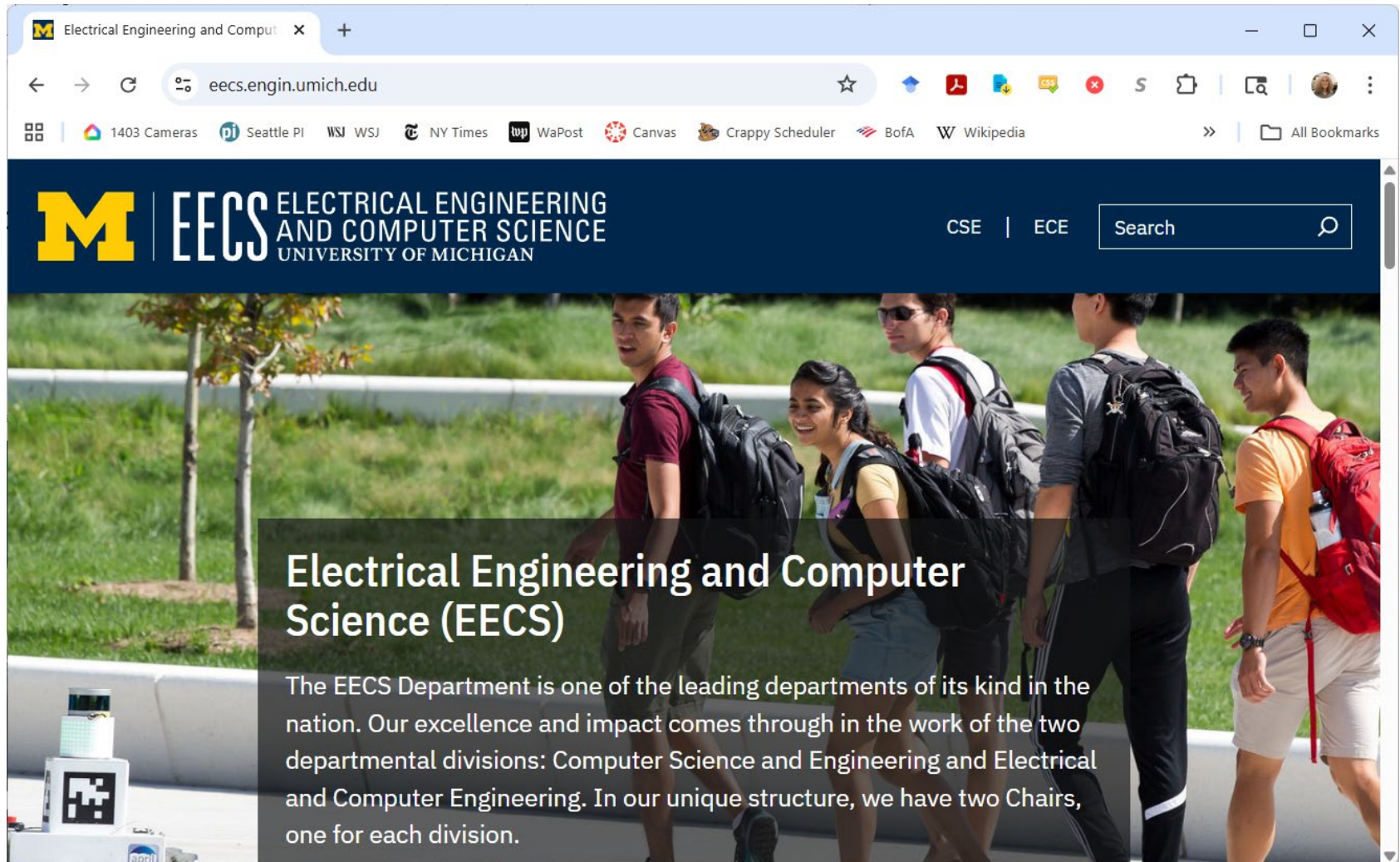
agenda

1. details
2. HTTP redirects
3. LinuxTinyServer
4. opening ports to WSL2
5. bind(), listen() and accept()
6. the Talk() thread
7. a plugin interface
8. forms

What's going on with this redirect? It's obviously wrong.

```
tcsh-1% LinuxGetUrl http://web.eecs.umich.edu/  
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">  
<html><head>  
<title>301 Moved Permanently</title>  
</head><body>  
<h1>Moved Permanently</h1>  
<p>The document has moved <a  
href="https://sites.google.com/a/umich.edu/petersonlab/">here</a>.</p>  
<hr>  
<address>Apache/2.4.58 (Ubuntu) Server at web.eecs.umich.edu Port  
80</address>  
</body></html>  
tcsh-2%
```

Why does it seem to work if I type it into a browser?



if it works for you

You're probably using Chrome or Edge which hide the problem.

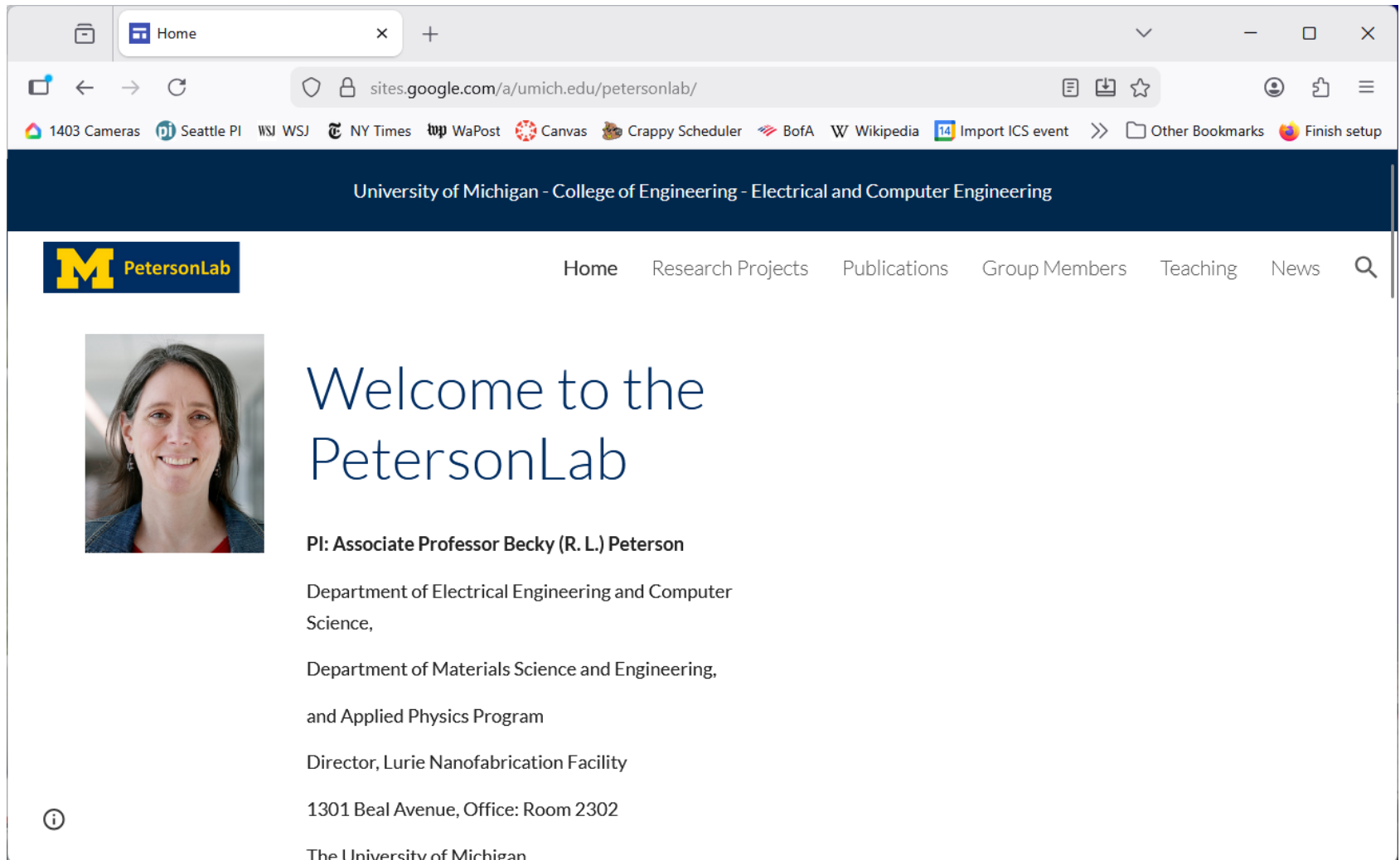
Chrome always tries the HTTPS version of any URL first.

Edge maintains a curated lists of site likely to support HTTPS and if the domain is on the list, tries HTTPS first.

(Apparently, the thinking is that lots of old links have never been updated, even if the server was.)

So, neither Chrome nor Edge ever notices the faulty redirect.

But if you use Firefox, this is what you'd see.



when crawling

You might decide to do what Chrome does, always trying HTTPS first.

If it doesn't support HTTPS, you might decide to discard the link.

agenda

1. details
2. HTTP redirects
3. **LinuxTinyServer**
4. opening a port to WSL2
5. `bind( )`, `listen( )` and `accept( )`
6. the `Talk( )` thread
7. a plugin interface
8. forms

A simple web server

You need a web server front end for your engine.

Two parts:

1. The HTTP server.
2. A plugin that can work with a form.

LinuxTinyServer

A very minimal web server for Linux.

1. It begins listening 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.  (But the autograder will
// not test this feature.)

// Usage:  LinuxTinyServer port rootdirectory

// Compile with g++ -pthread LinuxTinyServer.cpp -o LinuxTinyServer
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.

```
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. details
2. HTTP redirects
3. LinuxTinyServer
4. opening a port to WSL2
5. bind(), listen() and accept()
6. the Talk() thread
7. a plugin interface
8. forms

opening a port to WSL2

By default, the Windows Defender Firewall blocks most inbound connect requests coming from outside the machine.

Also, ports in WSL2 are virtualized and by default, also cannot be accessed except as localhost.

opening a port to WSL2

Two steps:

1. Open the port in the Windows Defender Firewall to Windows applications.
2. Set the networking mode in WSL2 to mirrored, meaning the network interfaces available in Windows are mirrored in WSL2.

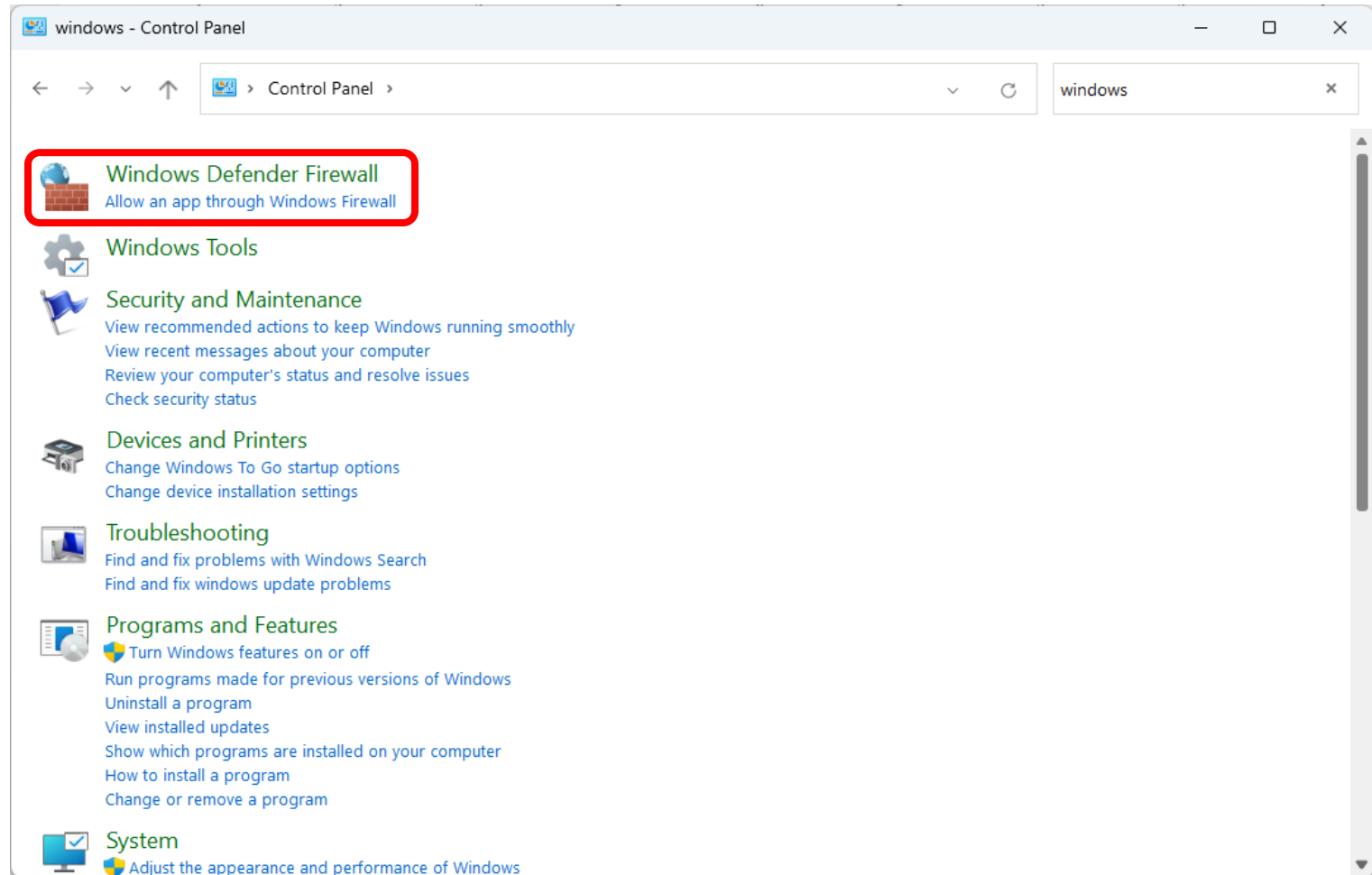
opening a port to WSL2

Two steps:

1. Open the port in the Windows Defender Firewall to Windows applications.
2. Set the networking mode in WSL2 to mirrored, meaning the network interfaces available in Windows are mirrored in WSL2.

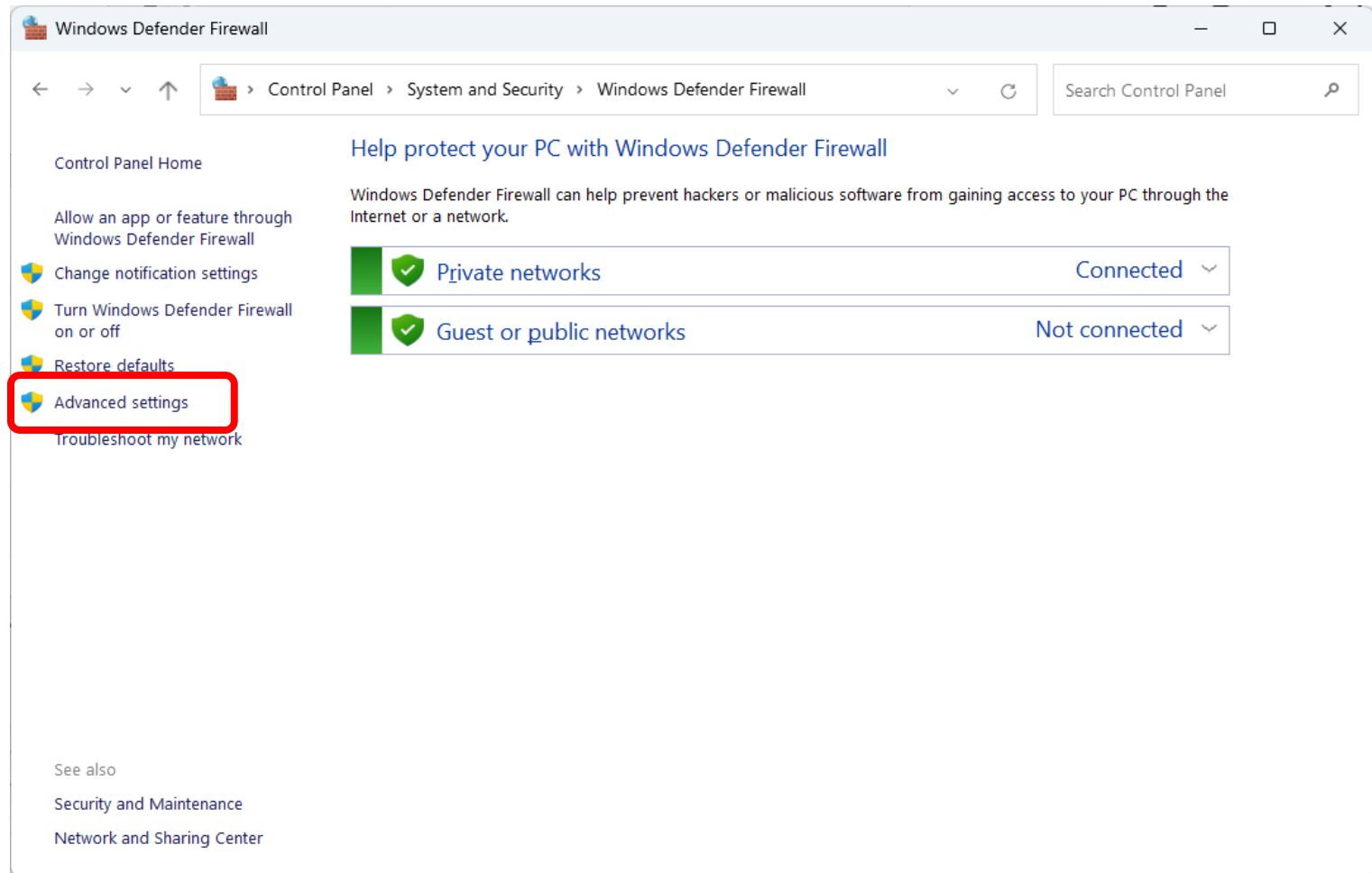
Windows Defender Firewall

1. It's in the control panel.



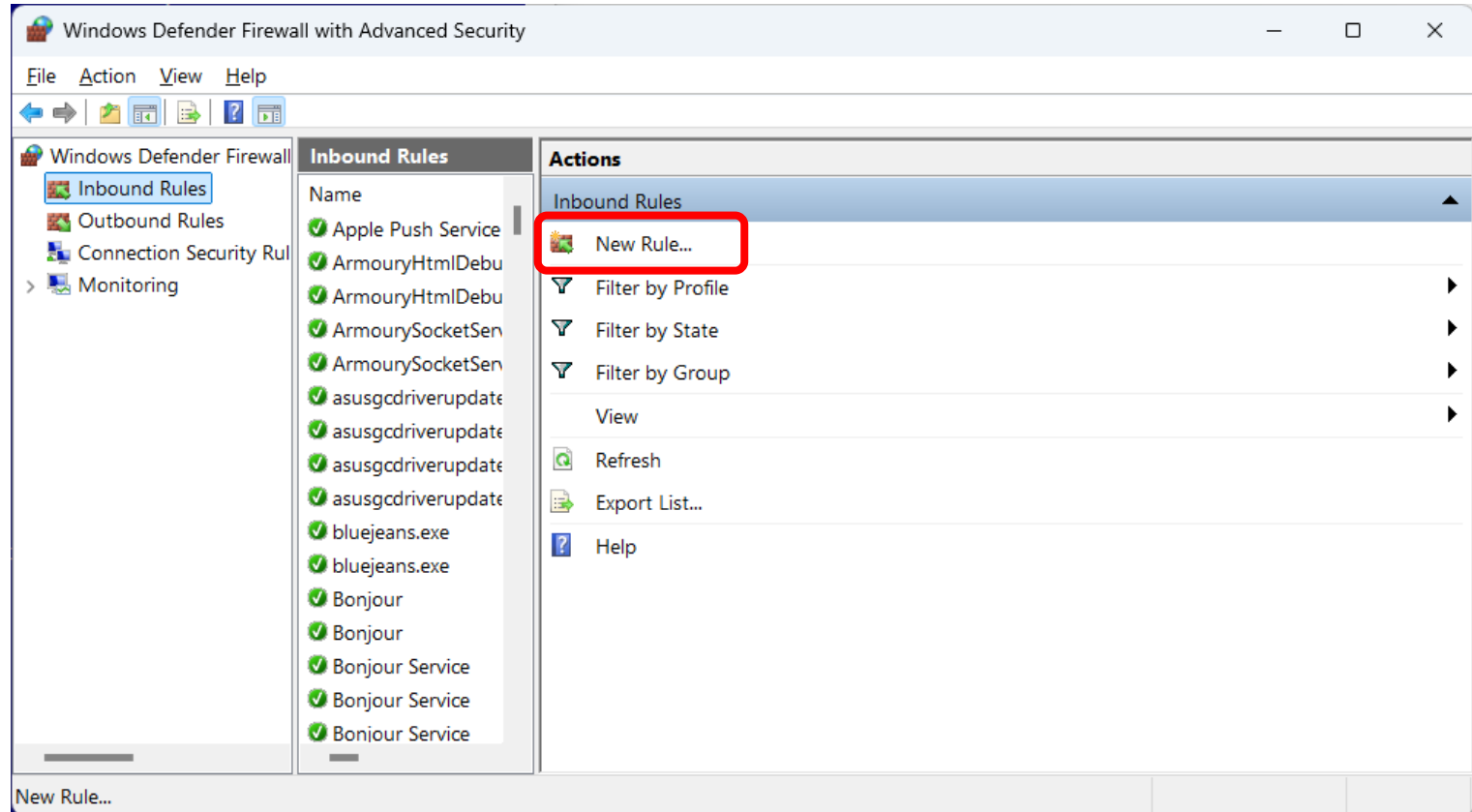
Windows Defender Firewall

2. Go to Advanced Settings.



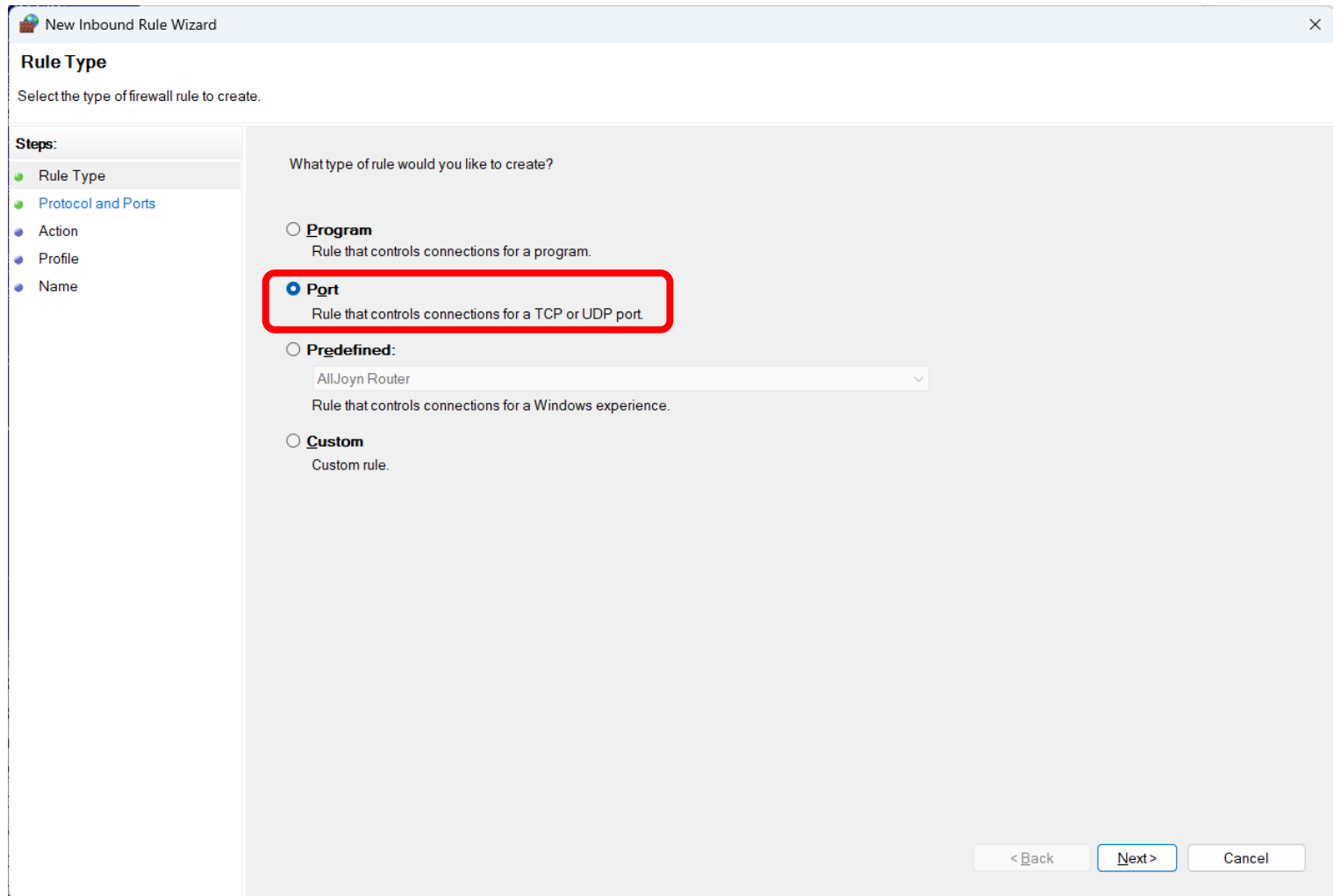
Windows Defender Firewall

3. Create a new rule.



Windows Defender Firewall

4. The new rule will open a port for inbound connections.



Windows Defender Firewall

5. The new rule will apply to TCP port 80 (the usual HTTP port).

New Inbound Rule Wizard

Protocol and Ports

Specify the protocols and ports to which this rule applies.

Steps:

- Rule Type
- Protocol and Ports
- Action
- Profile
- Name

Does this rule apply to TCP or UDP?

☒ TCP

☐ UDP

Does this rule apply to all local ports or specific local ports?

☐ All local ports

☒ Specific local ports:

Example: 80, 443, 5000-5010

< Back Next > Cancel

Windows Defender Firewall

6. The new rule will apply to any TCP port 80 you specify.

New Inbound Rule Wizard

Protocol and Ports

Specify the protocols and ports to which this rule applies.

Steps:

- Rule Type
- Protocol and Ports
- Action
- Profile
- Name

Does this rule apply to TCP or UDP?

☒ TCP

☐ UDP

Does this rule apply to all local ports or specific local ports?

☐ All local ports

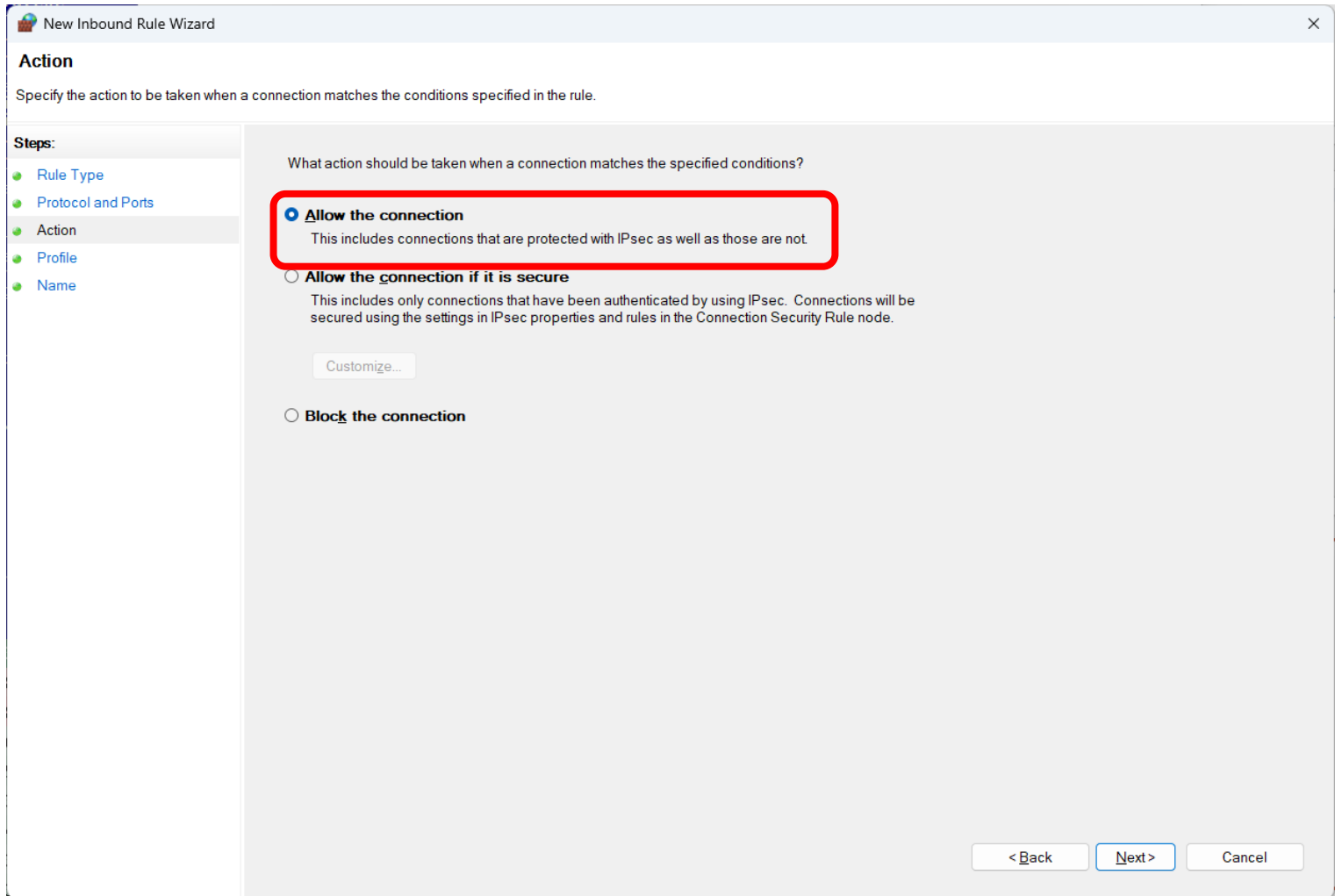
☒ Specific local ports:

Example: 80, 443, 5000-5010

< Back Next > Cancel

Windows Defender Firewall

7. Allow inbound connections to this port.



The screenshot shows the 'New Inbound Rule Wizard' window. The title bar reads 'New Inbound Rule Wizard'. The main heading is 'Action', with a subtitle 'Specify the action to be taken when a connection matches the conditions specified in the rule.' On the left, a 'Steps:' sidebar lists: Rule Type, Protocol and Ports, Action (highlighted), Profile, and Name. The main area asks 'What action should be taken when a connection matches the specified conditions?'. Three options are listed: 'Allow the connection' (selected and highlighted with a red rectangle), 'Allow the connection if it is secure', and 'Block the connection'. The 'Allow the connection' option includes the text 'This includes connections that are protected with IPsec as well as those are not'. The 'Allow the connection if it is secure' option includes the text 'This includes only connections that have been authenticated by using IPsec. Connections will be secured using the settings in IPsec properties and rules in the Connection Security Rule node.' and a 'Customize...' button. At the bottom right are buttons for '< Back', 'Next >', and 'Cancel'.

Action
Specify the action to be taken when a connection matches the conditions specified in the rule.

Steps:

- Rule Type
- Protocol and Ports
- Action**
- Profile
- Name

What action should be taken when a connection matches the specified conditions?

☒ **Allow the connection**
This includes connections that are protected with IPsec as well as those are not.

☐ **Allow the connection if it is secure**
This includes only connections that have been authenticated by using IPsec. Connections will be secured using the settings in IPsec properties and rules in the Connection Security Rule node.

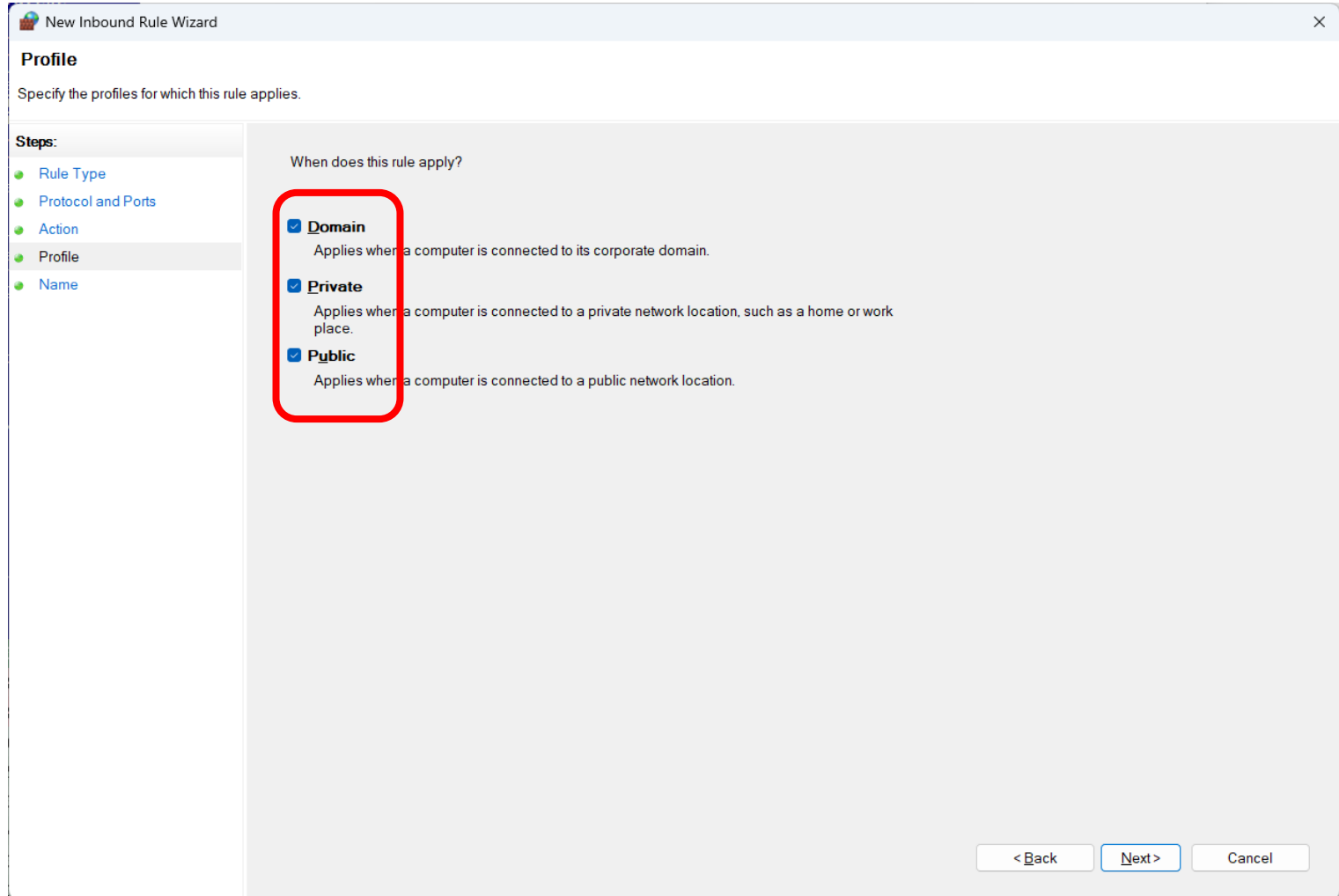
Customize...

☐ **Block the connection**

< Back Next > Cancel

Windows Defender Firewall

8. Applies anywhere.



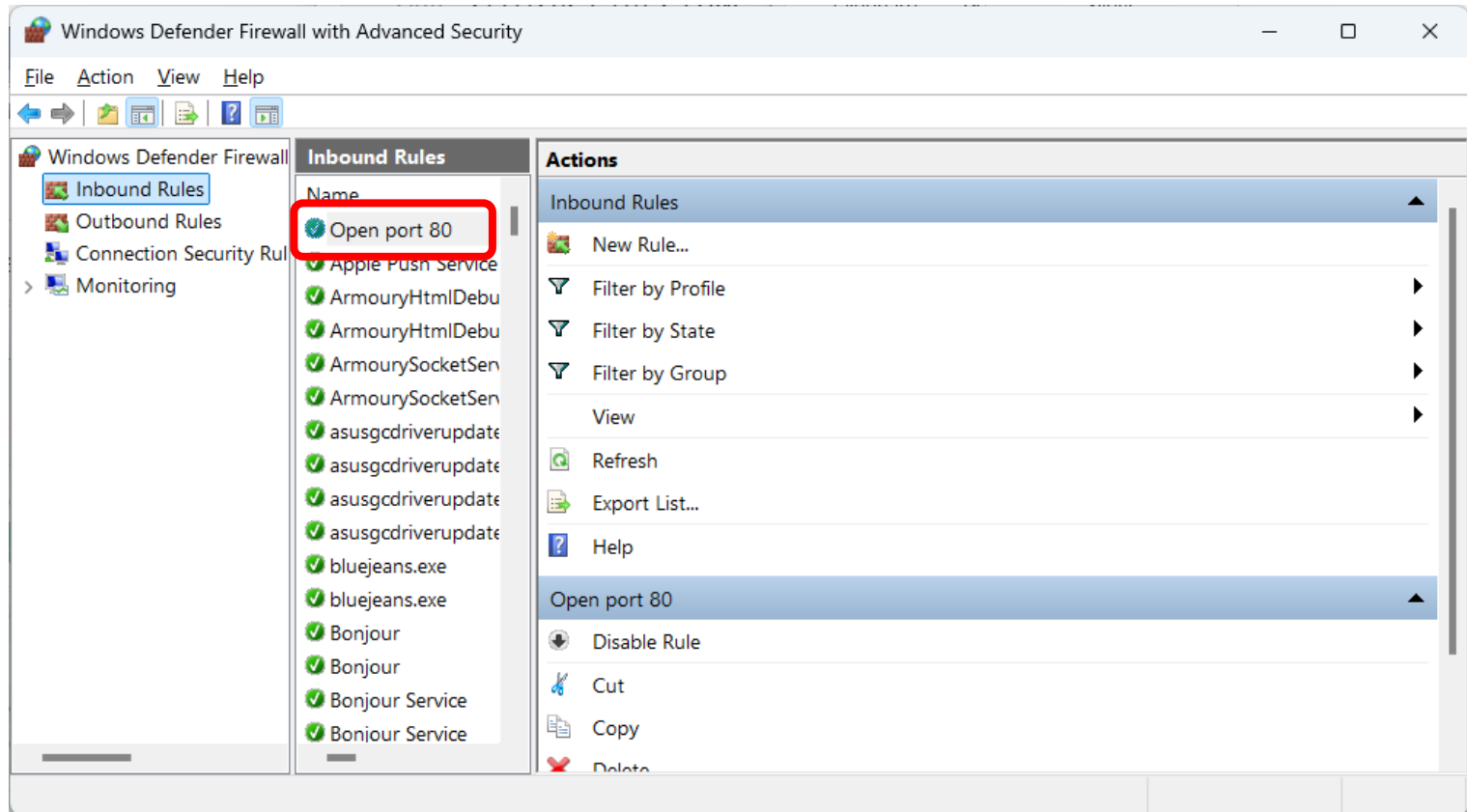
Windows Defender Firewall

9. Give it a name and click Finish.

The screenshot shows the 'New Inbound Rule Wizard' window. The title bar reads 'New Inbound Rule Wizard'. The main heading is 'Name', with the instruction 'Specify the name and description of this rule.' Below this, a 'Steps:' sidebar on the left lists: Rule Type, Protocol and Ports, Action, Profile, and Name (which is highlighted). The main area contains a 'Name:' label followed by a text box containing 'Open port 80'. Below this is a 'Description (optional):' label followed by a larger text box. At the bottom right, there are three buttons: '< Back', 'Finish', and 'Cancel'. The 'Finish' button is highlighted with a red rectangle.

Windows Defender Firewall

10. Now have a new rule opening port 80 to Windows applications.



opening a port to WSL2

Two steps:

1. Open the port in the Windows Defender Firewall to Windows applications.
2. Set the networking mode in WSL2 to mirrored, meaning the network interfaces available in Windows are mirrored in WSL2.

~/.wslconfig

Create a ~/.wslconfig file with this content to allow WSL2 apps to access the Windows network interfaces.

```
[wsl2]  
networkingMode=mirrored
```

privileged ports

In all Linux systems, ports numbered 1023 and below are **privileged ports**.

Only processes running with **root privileges** (superuser access) can bind to these ports.

Ensures that if you connect to a standard service like HTTP (port 80) or HTTPS (port 443), you can be reasonably confident you are communicating with the legitimate, system-controlled application, and not a malicious program set up by an unprivileged user.

agenda

1. details
2. HTTP redirects
3. LinuxTinyServer
4. opening a port to WSL2
5. `bind( )`, `listen( )` and `accept( )`
6. the `Talk( )` thread
7. a plugin interface
8. forms

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;
};
```

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 a *new* socket file descriptor for talking to the client.

The sockaddr that's returned 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 );  
}
```

agenda

1. details
2. LinuxTinyServer
3. opening ports to WSL2
4. `bind( )`, `listen( )` and `accept( )`
5. the `Talk( )` thread
6. a plugin interface
7. forms

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.
}
```

agenda

1. details
2. LinuxTinyServer
3. opening ports to WSL2
4. bind(), listen() and accept()
5. the Talk() thread
6. a plugin interface
7. forms

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 plugin object

A plugin inherits from this in plugin.h.

```
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.
```

The initial value is null.

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

the plugin interface

Here's how
LinuxTinyServer
checks for a plugin
and whether it's a
magic path.

If it's magic, let the
plugin process the
request and return
whatever it
responds.

```
// Check to see if there's a plugin and, if there is,  
// whether this is a magic path intercepted by the plugin.  
  
if ( Plugin && Plugin->MagicPath( r.Path ) )  
{  
    string request( buffer, bytes );  
    string response = Plugin->ProcessRequest( request );  
    cout << response << endl;  
    send( ts, response.c_str( ), response.size( ), 0 );  
    close( ts );  
    return nullptr;  
}
```

agenda

1. details
2. LinuxTinyServer
3. opening ports to WSL2
4. `bind( )`, `listen( )` and `accept( )`
5. the `Talk( )` thread
6. a plugin interface
7. forms

forms

Several ways of doing this.

1. GET
2. POST
3. REST using JSON

```

<!DOCTYPE html>
<html>
  <head>
    <title>Simple Form using GET</title>
    <link href="SimpleForm.css" rel="stylesheet">
  </head>

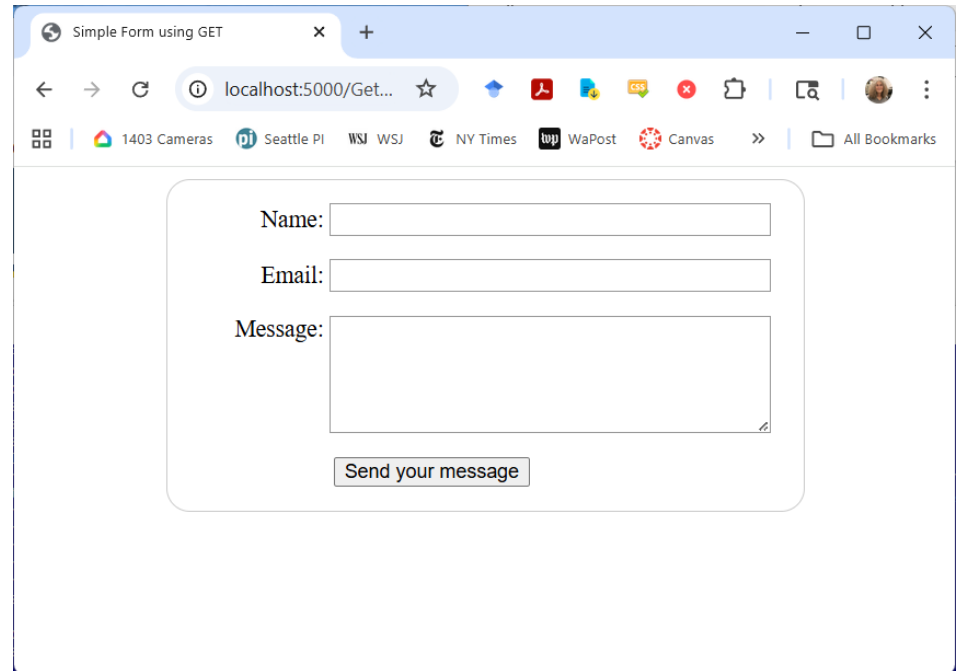
  <body>
    <form action="/GetForm.htm" method="get">
      <div>
        <label for="name">Name:</label>
        <input type="text" id="name" name="user_name" />
      </div>

      <div>
        <label for="mail">Email:</label>
        <input type="email" id="mail" name="user_email" />
      </div>

      <div>
        <label for="msg">Message:</label>
        <textarea id="msg" name="user_message"></textarea>
      </div>

      <div class="button">
        <button type="submit">Send your message</button>
      </div>
    </form>
  </body>
</html>

```



The screenshot shows a web browser window with the title "Simple Form using GET". The address bar shows "localhost:5000/Get...". The browser's toolbar includes navigation buttons, a search bar, and several extension icons. The main content area displays a form with the following elements:

- Name:** A text input field.
- Email:** An email input field.
- Message:** A text area for a longer message.
- Send your message:** A submit button.

Typing this and pressing
Send sends a GET
message to the server
with the form information
passed as ? query string.

Simple Form using GET

localhost:5000/Get...

Name: Nicole Hamilton

Email: nicole@nicolehamilton.com

Message: Hello world

Send your message

`http://localhost:5000/GetForm.htm?user_name=Nicole+Hamilton&user_email=nicole%40nicolehamilton.com&user_message=Hello+world`

LinuxTinyServer sees this.

```
GET /GetForm.htm?user_name=Nicole+Hamilton&user_email=nicole%40nicolehamilton.com&user_message=Hello+world HTTP/1.1
Host: localhost:5000
Connection: keep-alive
:
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/139.0.0.0 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
:
Referer: http://localhost:5000/GetForm.htm
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9
```

Since that's not a filename and I haven't defined a plugin, it fails as a 404.

```
Path does not exist.  
HTTP/1.1 404 Not Found  
Content-Length: 0  
Connection: close
```

But obviously, we could intercept that and make it do something useful.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Simple Form using POST</title>
    <link href="SimpleForm.css" rel="stylesheet">
  </head>

  <body>
    <form action="/PostForm.htm" method="post">
      <div>
        <label for="name">Name:</label>
        <input type="text" id="name" name="user_name" />
      </div>

      <div>
        <label for="mail">Email:</label>
        <input type="email" id="mail" name="user_email" />
      </div>

      <div>
        <label for="msg">Message:</label>
        <textarea id="msg" name="user_message"></textarea>
      </div>

      <div class="button">
        <button type="submit">Send your message</button>
      </div>
    </form>
  </body>
</html>

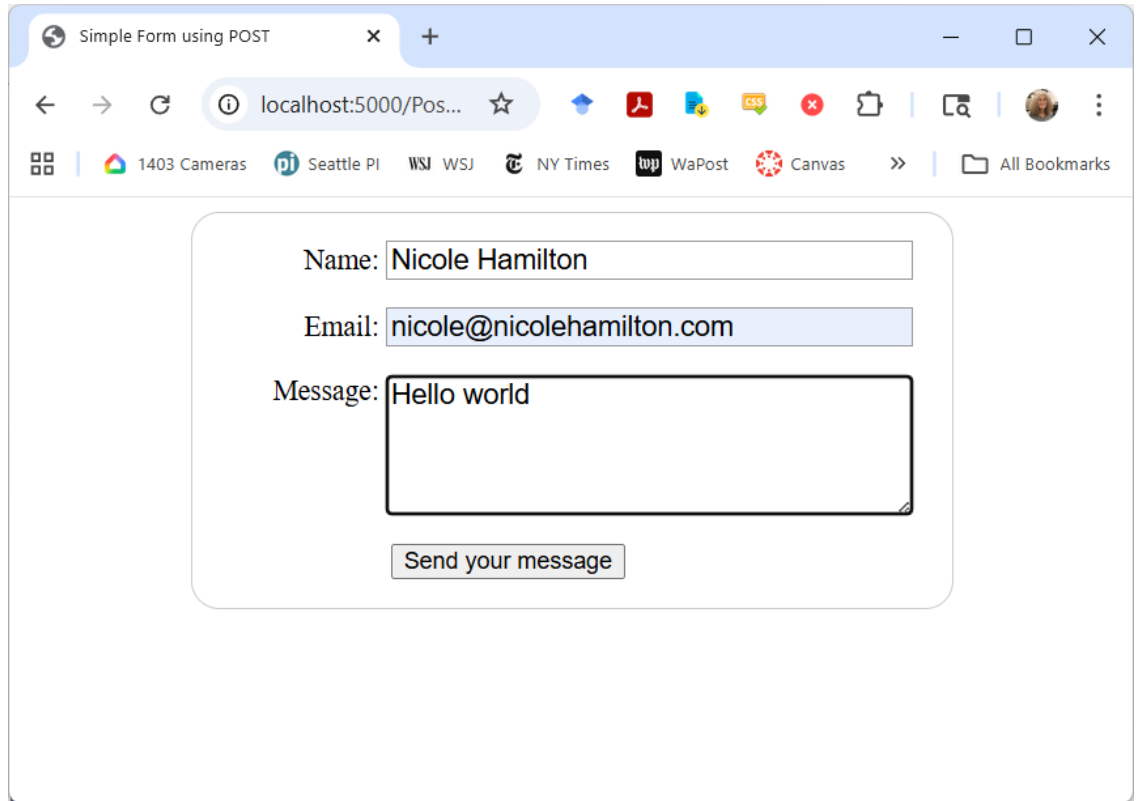
```

The screenshot shows a web browser window with the title 'Simple Form using POST'. The address bar shows 'localhost:5000/Pos...'. The browser's toolbar includes navigation buttons, a search bar, and several open tabs. The main content area displays a form with the following elements:

- Name:** A text input field.
- Email:** An email input field.
- Message:** A text area for a longer message.
- Send your message:** A submit button.

Typing this and pressing
Send sends a POST
message to the server
with the form information
passed as content.

The URL is unchanged.



The screenshot shows a web browser window with the title "Simple Form using POST". The address bar displays "localhost:5000/Pos...". The browser's bookmark bar includes "1403 Cameras", "Seattle PI", "WSJ", "NY Times", "WaPost", and "Canvas". The form itself is contained within a rounded rectangle and includes three input fields: "Name:" with the value "Nicole Hamilton", "Email:" with the value "nicole@nicolehamilton.com", and "Message:" with the value "Hello world". Below these fields is a button labeled "Send your message".

LinuxTinyServer sees this.

```
POST /PostForm.htm HTTP/1.1
Host: localhost:5000
Connection: keep-alive
Content-Length: 89
:
Origin: http://localhost:5000
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/139.0.0.0 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image
/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
:
Referer: http://localhost:5000/PostForm.htm
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9

user_name=Nicole+Hamilton&user_email=nicole%40nicolehamilton.com&user_
message=Hello+world
```

It doesn't understand POST and I haven't defined a plugin, it also fails.

```
Path does not exist.  
HTTP/1.1 404 Not Found  
Content-Length: 0  
Connection: close
```

But obviously, we could intercept this also and make it do something useful.

280 OfficeHours example

Example: The 280 Office Hours 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( )
    {
    }
};
```

280 OfficeHours example

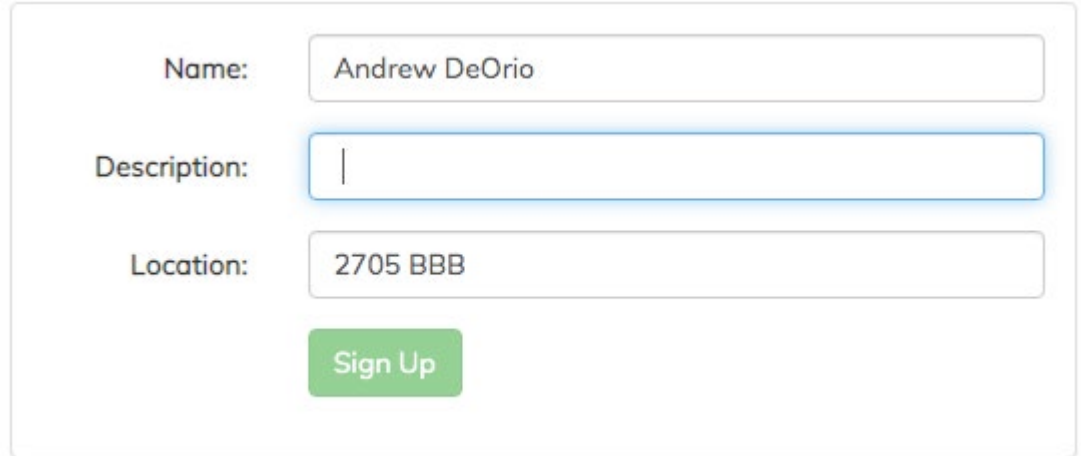
A Javascript script communicates with the host by using GET, POST, and DELETE requests.

Data is passed in JSON format.

```
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( )
    {
    }
};
```

The HTML form has named elements that can be accessed from Javascript.

When you enter data, that's POST'ed to the host as JSON.



Name: Andrew DeOrio

Description: |

Location: 2705 BBB

Sign Up

Browser sends:

```
POST /api/queue/tail/ HTTP/1.1
Host: localhost
Content-Type: application/json;
charset=utf-8
Content-Length: 59
```

```
{
  "username": "awdeorio",
  "location": "2705 BBB"
}
```

Webserver responds:

```
HTTP/1.1 201 Created
Content-Type: application/json;
charset=utf-8
Content-Length: 78
```

```
{
  "location": "2705 BBB",
  "position": 1,
  "username": "awdeorio"
}
```