**Internet & Networking**

**Network Protocols - HTTP & HTTPS**

**Report on HTTP and HTTPS Transactions**

**1. Overview of HTTP and HTTPS Transactions**

HTTP (Hypertext Transfer Protocol)

**Purpose**: HTTP is a protocol used for transferring hypertext requests and information on the internet.

**Data Transmission**: Data is transmitted in plaintext.

**Security:** Lacks encryption, making data vulnerable to interception and manipulation.

**Common Uses**: Simple web pages, non-sensitive data exchange.

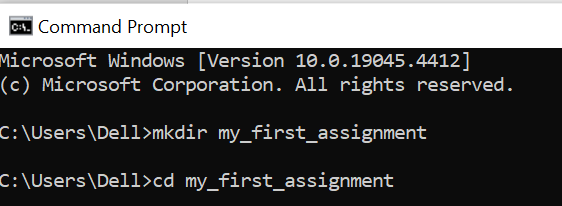
HTTPS (Hypertext Transfer Protocol Secure)

**Purpose**: HTTPS is an extension of HTTP designed to secure data transmitted over the internet.

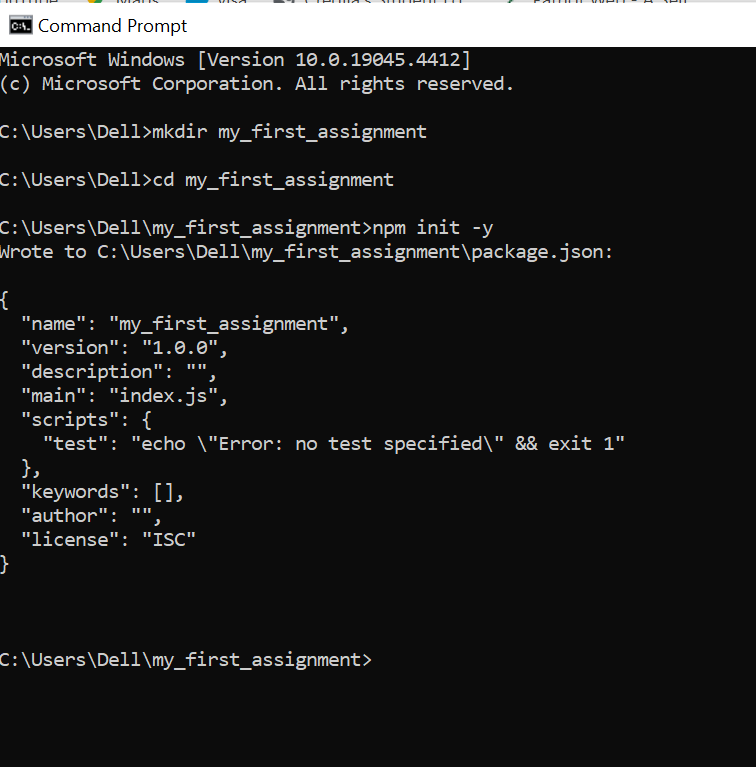
**Data Transmission**: Data is encrypted using SSL/TLS.

**Security**: Provides encryption, data integrity, and authentication, making data transmission secure.

**2.** We will be using Node.js with the Express framework, as it is quick and easy for creating a basic web server.

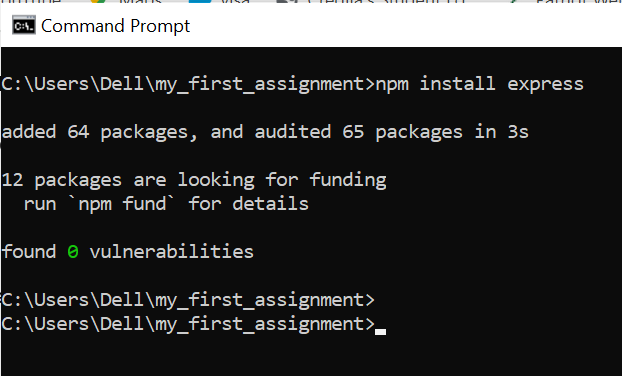


Create a New Node.js Project



Since we created new file we will be getting package.json by default.

Installing the Express framework it helps in creating web applications



We will be creating a new file named server.js with the following code in it.

const express = require('express');

const app = express();

const port = 3000;

// Serve static files from the 'public' directory

app.use(express.static('public'));

// Define a route to serve content

app.get('/', (req, res) => {

res.send('Hello, World!');

});

// Start the server

app.listen(port, () => {

console.log(`Server is running at http://localhost:${port}`);

});

Now Create a new folder named Public which will be holding static files

Now we will add a HTML file (index.html) inside the public folder

<!DOCTYPE html>

<html>

<head>

<title>My\_first\_assignment</title>

</head>

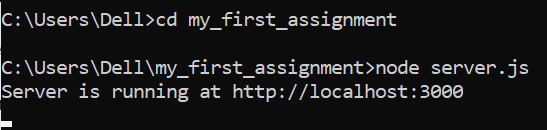
<body>

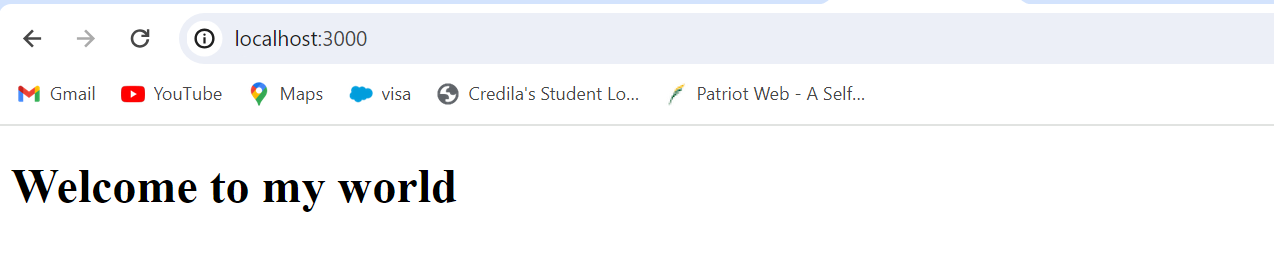
<h1>Welcome to my world</h1>

</body>

</html>

Now we will run node server.js in my\_first\_assignment directory so that we can see where it runs





HTTPS Implementation: Upgrade the server to serve content over HTTPS\

Form HTTP To HTTPS we need some openssl self signed ceritificates

We will run these code in directory

openssl genrsa -out server.key 2048

openssl req -new -key server.key -out server.csr

openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt

After running we will be getting these files

server.key: private key.

server.csr: certificate signing request (not used in this guide).

server.crt: self-signed certificate.

Now we need to update server.js code since we are moving from http to https

const express = require('express');

const https = require('https');

const fs = require('fs');

const path = require('path');

const app = express();

const port = 3000;

// SSL/TLS certificate and key

const options = {

key: fs.readFileSync(path.join(\_\_dirname, 'server.key')),

cert: fs.readFileSync(path.join(\_\_dirname, 'server.crt'))

};

// Serve static files from the 'public' directory

app.use(express.static('public'));

// Define a route to serve content

app.get('/', (req, res) => {

res.send('Hello, Secure World!');

});

// Start the HTTPS server

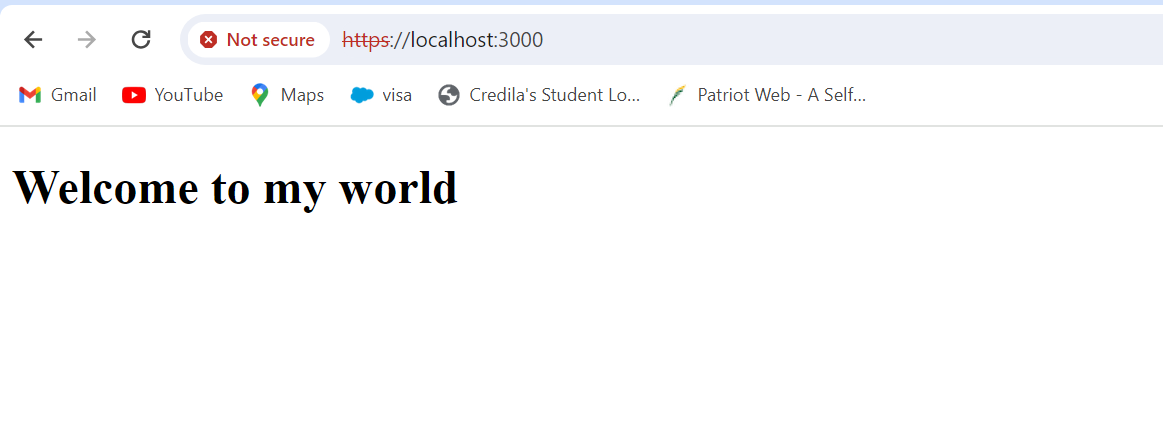
https.createServer(options, app).listen(port, () => {

console.log(`Secure server is running at https://localhost:${port}`);

});

Now again running node server.js shows where the site runs

Since the certificate is self-signed, browser will warn us saying that the connection is not secure.



**3. Analysis of Security Differences Between HTTP and HTTPS Traffic**

**Encryption**:

**HTTP**: No encryption; data is sent in plaintext, making it readable by anyone who intercepts it.

**HTTPS**: Uses SSL/TLS to encrypt data, ensuring that intercepted data cannot be read.

**Data Integrity**:

**HTTP**: No checks for data integrity; data can be altered during transmission without detection.

**HTTPS**: Ensures data integrity using cryptographic hash functions; any alteration of data is detected.

**Authentication**:

**HTTP**: No inherent authentication mechanism; vulnerable to spoofing attacks.

**HTTPS**: Uses SSL/TLS certificates to authenticate the server and optionally the client; clients can verify they are communicating with the legitimate server.

**Vulnerabilities**:

**HTTP**: Susceptible to eavesdropping, MITM attacks, and data tampering.

**HTTPS**: Protects against eavesdropping and MITM attacks through encryption and authentication; ensures data integrity.

**Performance**:

**HTTP**: Slightly faster due to lack of encryption/decryption overhead.

**HTTPS**: Slightly slower due to SSL/TLS overhead, though modern hardware minimizes this impact.

**SEO and Browser Support**:

**HTTP**: Less preferred by search engines; modern browsers mark sites as "Not Secure."

**HTTPS**: Preferred by search engines, leading to better SEO rankings; modern browsers display a padlock icon, indicating a secure connection.

**Practical Implications**

**Login Forms**:

**HTTP**: Credentials are transmitted in plaintext, easily intercepted.

**HTTPS**: Credentials are encrypted, protecting them from interception.

**Online Shopping**:

**HTTP**: Credit card information is vulnerable to theft.

**HTTPS**: Credit card information is encrypted, significantly reducing the risk of theft.

**Conclusion**

HTTPS provides significant security enhancements over HTTP, including encryption, data integrity, and authentication. These features protect sensitive data from interception and tampering, ensuring secure communication. Despite the slight performance overhead, the benefits of HTTPS make it essential for any application handling sensitive information.

**DNS (Domain Name System)**

**1. A report documenting the DNS query path, including the tools and commands used.**

**Introduction**

A DNS query path report documents the steps and tools used to trace the path of a DNS query from the client to the authoritative DNS server. I have made a report uses the `dig` tool to trace the DNS query path for `google.com`.

**Tools and Commands**

1. **Tool**: `dig` (Domain Information Groper)

- `dig` is a command-line tool used for querying DNS name servers. It provides detailed information about DNS queries and responses.

**Executing the DNS Query**

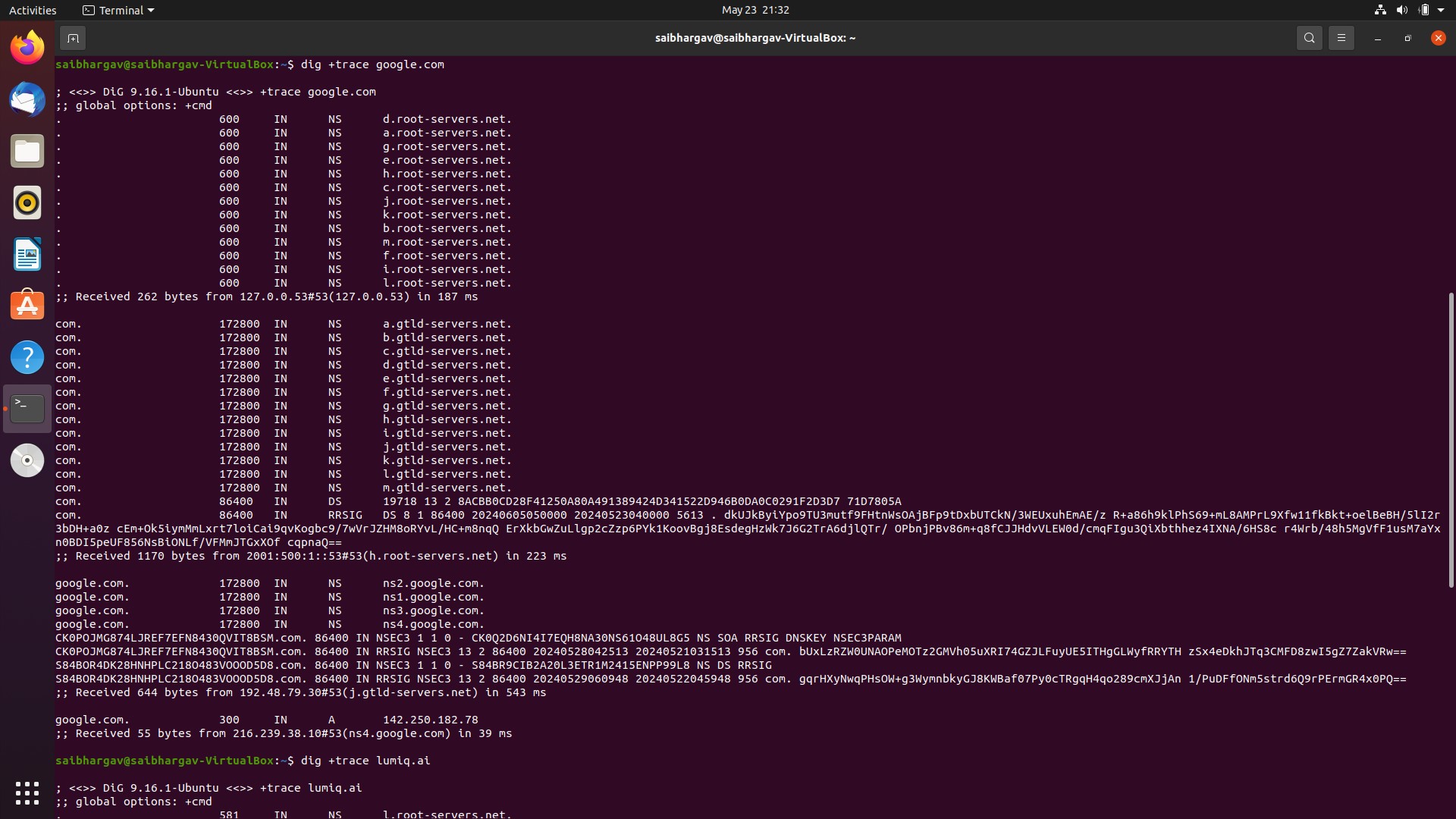
1. Open the Linux terminal

2. Run the `dig` command with the `+trace` option to trace the DNS query path for `google.com`:

dig +trace google.com

Analysis of the DNS Query Path

The `dig +trace` command provides a step-by-step resolution path, starting from the root name servers down to the authoritative name servers for `google.com`. Below is an output and explanation.



**Explanation**:

1. **Root Servers**: The query starts at the root DNS servers, which provide information about the top-level domain (TLD) name servers for `.com`.

2. **TLD Servers**: The `.com` TLD servers provide information about the authoritative name servers for `google.com`.

3. **Authoritative Servers:** The authoritative name servers for `google.com` provide the final IP addresses for `google.com`.

**2.Demonstration of set up of DNS server**

I am using BIND (Berkeley Internet Name Domain) to set up DNS server.

**Install BIND**

sudo apt update

sudo apt install bind9 bind9utils bind9-doc

After installation, check thatDNS server service is running

sudo systemctl start bind9

**3. DNS Server Configuration: Configure the hosted DNS server to resolve specified domain names to predetermined IP addresses.**

I have taken google.com as example to set up.

**Configure BIND**

it is located at /etc/bind/named.conf

**Since we have taken google as example, we will define it in configuration file**

zone "google.com" {

type master;

file "/etc/bind/db.google.com";

};

**Creating the zone file for google.com**.

sudo nano /etc/bind/db.google.com

**Add the following content to the zone file**

$TTL 86400

@ IN SOA ns1.google.com. admin.google.com. (

2024052201; Serial

3600 ; Refresh

1800 ; Retry

604800 ; Expire

86400 ) ; Minimum TTL

@ IN NS ns1.google.com.

@ IN NS ns2.google.com.

ns1 IN A 192.168.1.10 ;

ns2 IN A 172.29.192.1 ;

@ IN A 142.250.74.142

www IN A 142.250.74.142

**Restart BIND Service**

sudo systemctl restart bind9

**Test the DNS Server**

dig google.com



**Public and Private Networks**

**OWASP Vulnerabilities**

**A document containing explanations of each OWASP Top Ten Vulnerability**

1. Injection

Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

2. Broken Authentication

Broken authentication refers to vulnerabilities that allow attackers to compromise passwords, keys, or session tokens, or to exploit implementation flaws to assume other users' identities temporarily or permanently.

3. Sensitive Data Exposure

Sensitive data exposure occurs when applications do not adequately protect sensitive information, such as financial data, health records, and personal identifiers, during transmission or storage.

4. XML External Entities (XXE)

XXE vulnerabilities occur when XML parsers process external entities within XML documents. Attackers can exploit these entities to expose internal files, internal network ports, and execute remote code.

5. Broken Access Control

Broken access control vulnerabilities occur when restrictions on authenticated users are not properly enforced, allowing attackers to act outside their intended permissions.

6. Security Misconfiguration

Security misconfiguration occurs when security settings are not defined, implemented, or maintained properly. This includes default configurations, incomplete configurations, open cloud storage, and unnecessary services enabled.

7. Cross-Site Scripting (XSS)

XSS vulnerabilities occur when an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript.

8. Insecure Deserialization

Insecure deserialization occurs when untrusted data is used to abuse the logic of an application, inflict denial of service (DoS) attacks, or execute arbitrary code upon deserialization.

9. Using Components with Known Vulnerabilities

This vulnerability arises when applications use libraries, frameworks, and other software modules with known vulnerabilities. If these components are exploited, it can lead to significant consequences.

10. Insufficient Logging and Monitoring

Insufficient logging and monitoring combined with missing or ineffective integration with incident response allow attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data.

**A set of brief summaries highlighting the key aspects of each vulnerability, including potential impacts and risks.**

1. Injection

Key Aspects: Untrusted data sent to an interpreter as part of a command or query, leading to unintended commands or unauthorized data access.

Potential Impacts: Data theft, corruption, loss, denial of service, complete host takeover.

Risks: High; can compromise entire systems and data integrity.

2. Broken Authentication

Key Aspects: Vulnerabilities in authentication mechanisms allowing attackers to compromise credentials and assume other users' identities.

Potential Impacts: Credential theft, unauthorized account access, identity theft.

Risks: High; leads to account takeovers and unauthorized access to sensitive data.

3. Sensitive Data Exposure

Key Aspects: Inadequate protection of sensitive information during transmission or storage.

Potential Impacts: Data breaches, legal consequences, loss of customer trust.

Risks: High; especially critical in finance and healthcare sectors handling sensitive data.

4. XML External Entities (XXE)

Key Aspects: Exploitation of XML parsers processing external entities, leading to exposure of internal files and remote code execution.

Potential Impacts: Disclosure of internal files, remote code execution, denial of service.

Risks: Medium to high; depending on the extent of data exposed and use of XML.

5. Broken Access Control

Key Aspects: Improper enforcement of restrictions on authenticated users, allowing unauthorized actions.

Potential Impacts: Unauthorized access to data and functions, data modification or deletion.

Risks: High; can lead to unauthorized actions within the application.

6. Security Misconfiguration

Key Aspects: Inadequate definition, implementation, or maintenance of security settings.

Potential Impacts: Data exposure, unauthorized access, system compromise.

Risks: Medium to high; can result in a range of issues from data exposure to system takeover.

7. Cross-Site Scripting (XSS)

Key Aspects: Inclusion of untrusted data in web pages without proper validation or escaping, leading to execution of malicious scripts.

Potential Impacts: Session hijacking, defacement, redirection to malicious sites.

Risks: Medium; impacts user trust and application functionality.

8. Insecure Deserialization

Key Aspects: Untrusted data used during deserialization, leading to abuse of application logic or remote code execution.

Potential Impacts: Remote code execution, data manipulation, denial of service.

Risks: Medium to high; based on the application's serialization use.

9. Using Components with Known Vulnerabilities

Key Aspects: Use of libraries, frameworks, or other software modules with known vulnerabilities.

Potential Impacts: Data breaches, system compromise.

Risks: High; exploitation through documented vulnerabilities.

10. Insufficient Logging and Monitoring

Key Aspects: Lack of effective logging and monitoring, combined with inadequate incident response integration.

Potential Impacts: Prolonged undetected attacks, delayed security incident responses.

Risks: Medium to high; can exacerbate impacts by delaying detection and response to breaches.