# Contents

## 1. Introduction

This project involves setting up a system that starts with the installation of necessary applications, preventing the need for repeated installations. It conducts an anonymity check of the network connection, promptly alerting if it is non-anonymous and revealing the spoofed country name if anonymous to retrieve server details and execute commands which includes Whois and Nmap under masked IP through NIPE (a.k.a TOR) via a remote server connection from the local server. Finally, it saves the collected data into local sever directory, sub-directories and files and maintains a log for auditing data collection activities.

## 2. Project Objective:

### 2.1 Scope 1: Script Breakdown and Explanation

The purpose of this project lies in the creation of a bash script key feature with the flexibility in accepting user-specified scan targets, putting the user in control. A script breakdown will be provided in this report to provide details of the purpose and function of the command line of the script under section 3.

### 2.2 Scope 2: Network Research & Monitoring

Under section 4, we will be moving into scope 2 of the project, an analysis of the network traffic during the running of the script will be provided. Section 5 will provide research on HTTP protocol to provide understanding of its purpose, key features and the problem it aims to solve and the strengths and weaknesses in relation to the CIA triad. Section 6 will also be touching base on the suggested secure protocol HTTPS in great details on the design, mechanics and illustrative example of its behavior.

Based on what was research on section 6, the impact of the of https protocol against the existing risk in relation to the http protocol; along with the recommendations for risk mitigation of the weakness of the http protocol through https will be provided under section 7 and 8. Follow by the conclusion of the report under section 9.

### 2.3. Methodologies

In order to achieve aim and outcome of the project the use of screenshots from the created Bash shell scripts, PCAP file, wireshark and online research of relevant online sources were made to deliver the objectives.

## 3. Overview and Analysis of Project Scope 1's Script Created: recon.sh

### 3.1 Script Beginning (Lines 1-7):

```
1 #!/bin/bash
2
3 #This is the begining of the script to inform user that the script has started
4 #printout to user on what the script is doing.
5 echo 'Greetings, User.
6 Checking for required programs'
7 echo ' '
```

- This section marks the beginning of the script and informs the user that the script has started.
- It also prints a message explaining what the script is doing ("Checking for required programs").

### 3.2 Color Codes (Lines 9-16):

```
 9 #These colour codes to increase key command words' visibility
10 #colour codes assignment.
11 RED='\033[0;31m'
12 GRN='\033[0;32m'
13 YLW='\033[0;33m'
14 BGRN='\033[1;32m'
15 BCYN='\033[1;36m'
16 CLR='\033[0m'
```

- This section defines color codes for highlighting keywords in the script's output.
- These codes are assigned variables with names like RED, GRN (Green), YLW (Yellow), etc.

### 3.3 Installation Functions (Lines 18-44):

```
18 # Function to install NIPE, sshpass and geoip-bin
19 install_nipe() {
20         sudo apt update
21         echo "Cloning NIPE repository..."
22         sudo git clone https://github.com/htrgouvea/nipe.git && cd nipe
23
24         echo "Installing NIPE dependencies..."
25         cpanm --installdeps .
26         sudo cpan install Switch JSON LWP::UserAgent Config::Simple
27
28         echo "Installing NIPE..."
29         sudo perl nipe.pl install
30
31         echo "NIPE installation complete."
32 }
33
34 install_sshpass() {
35         sudo apt update
36         sudo apt install sshpass
37         echo "sshpass installation complete"
38 }
39
40 install_geoip() {
41         sudo apt update
42         sudo apt install geoip-bin
43         echo "geoip-bin installation complete"
44 }
```

- This section defines three functions:
    - install_nipe: This function updates package lists, clones the NIPE repository from GitHub, installs dependencies, and finally installs NIPE itself.
    - install_sshpass: This function updates package lists and installs the sshpass tool.
    - install_geoip: This function updates package lists and installs the geoip-bin package.

### 3.4 Dependency Checks (Lines 46-76):

```
46 #This checks for nipe.
47 echo ' '
48 NPL=$(find ~ -name nipe.pl)
49 if [ -z $NPL ]
50 then
51         echo -e "Nipe NOT found! Installing Nipe"
52         install_nipe
53 else
54         echo -e "Nipe ${GRN}detected${CLR}."
55 fi
56
57 #This checks for sshpass.
58 SSHP=$(ls /usr/bin | grep sshpass)
59 if [ -z $SSHP ]
60 then
61         echo -e "sshpass NOT found! Installing sshpass"
62         install_sshpass
63 else
64         echo -e "sshpass ${GRN}detected${CLR}."
65 fi
66
67 #This checks for geoip-bin.
68 GEOIP=$(find /usr/share -name 'geoip-bin')
69 if [ -z $GEOIP ]
70 then
71         echo -e "geoip-bin NOT found! Installing geoip-bin
72         install_geoip
73 else
74         echo -e "geoip-bin ${GRN}detected${CLR}."
75 fi
76 echo ' '
```

- This section checks if the required programs (Nipe, sshpass, and geoip-bin) are installed.
    - It uses find and ls commands to search for the program files.
    - If a program is not found, the script installs it using the corresponding installation function.
    - The script uses color codes (defined earlier) to highlight the program status (detected or not found).

### 3.5 IP Spoofing with NIPE (Lines 78-105):

```
78 #Installation checks ends
79 #IP spoofing with NIPE
80 #confirm that the spoof is active.
81 echo ' '
82 echo 'Proceeding to spoof IP...'
83 cd ~/nipe
84 #nipe can only be executed from the nipe directory so the script must proceed to the specific directory/folder where nipe.pl is located
85 sudo perl nipe.pl restart
86 #Restarts function initiates the service into an active state regardless of nipe's status.
87
88 STAT=$(sudo perl nipe.pl status | grep -i true)
89 IP=$(sudo perl nipe.pl status | grep -i ip | awk '{print$3}')
90 #nipe status check.
91 if [ -z "$STAT" ]
92 #Double quotation marks were used to prevent the "'[] were use to prevent excessive arguments"
93 #Also preventserror arising from spaces in the output
94 then
95         echo -e "Spoof ${RED}NOT${CLR} active!
96 Please check nipe installation and/or network connections and run this script again."
97         exit
98
99 else
100         echo -e "Spoof ${GRN}ACTIVE${CLR}. Masking your original IP Address."
101         echo "Spoofed IP Address: $IP"
102         geoiplookup "$IP"
103
104 fi
105 echo ' '
```

- This section initiates IP spoofing using NIPE.
    - It navigates to the nipe directory (assuming the script is run from outside the directory).
    - It then uses sudo perl nipe.pl restart to start the spoofing service.
    - The script checks the status of the spoof using sudo perl nipe.pl status.
        - If the spoof is not active, it displays an error message and exits the script.
    - Otherwise, it displays a success message and retrieves the spoofed IP address.
    - It then attempts to perform a geolocation lookup on the spoofed IP using geoiplookup.

### 3.6 Remote Server Connection (Lines 107-129):

```
107 #The commands below communicates with the remote server.
108 #For the purposes of this script we shall assume that the remote server already has whois and nmap.
109
110 echo -e "Please enter ${YLW}remote${CLR} User login"
111 read REMLOG
112 echo -e "Please enter ${YLW}remote${CLR} IP"
113 read REMIP
114 echo -e "Please enter ${YLW}remote${CLR} User password"
115 read REMPW
116 echo -e "Please provide ${RED}TARGET${CLR} IP/Domain to scan"
117 read VIP
118 echo ' '
119
120 #Automated scanning and saving of logs into a file
121
122 #The command below provides the user input of remote server IP address entered earlier, country and uptime.
123 echo 'Connecting to Remote Server...'
124 IPTRUE=$(sshpass -p "$REMPW" ssh "$REMLOG@$REMIP" 'curl -s ifconfig.co')
125 echo "IP Address: $IPTRUE"
126 whois "$IPTRUE" | grep -i country | sort | uniq
127 UPT=$(sshpass -p "$REMPW" ssh "$REMLOG@$REMIP" 'uptime')
128 echo "Uptime: $UPT"
129 echo ' '
```

- This section prompts the user for credentials (username, password) to connect to a remote server.
- It also prompts the user for the target IP/domain to be scanned.

### 3.7 Automated Scanning and Saving Logs (Lines 131-141):

```
131 #Remote server will scan the target for the user.
132 #whois is scanned first, followed by nmap.
133 #The respective scan outputs will be saved into seperate files
134 echo ' '
135 echo 'Scanning Target...'
136 echo "Saving whois data into $VIP-whois"
137 sshpass -p "$REMPW" ssh "$REMLOG@$REMIP" "whois $VIP >> $VIP-whois"
138
139 echo "Saving nmap data into $VIP-nmap"
140 sshpass -p "$REMPW" ssh "$REMLOG@$REMIP" "nmap $VIP -Pn -sV -oN $VIP-nmap"
141 echo ' '
```

- This section connects to the remote server using sshpass and performs the following actions:
    - Retrieves the IP address of the remote server using curl -s ifconfig.co.
        - curl: This is a command-line tool used for transferring data over computer networks.
        - -s: This is a flag for curl that tells it to operate silently, meaning it won't display any progress messages on the screen.
        - ifconfig.co: This is the web address of a service that provides your public IP address. When curl fetches the content from this website, it only retrieves the actual IP address and avoids any additional information displayed on the website itself.

- Performs a whois lookup on the remote server's IP and saves the output to a file named $VIP-whois (where $VIP is the target IP/domain).
- Runs an nmap scan on the target IP/domain with specific flags (-Pn, -sV, -oN $VIP-nmap) and saves the output to a file named $VIP-nmap.
    - -Pn: This flag tells Nmap to skip the host discovery phase. It assumes all given hosts are up.
    - -sV: This flag enables version detection. Nmap attempts to determine the version of the services running on open ports.
    - -oN: This flag specifies the output format as normal. The scan results will be saved to a plain text file.

### 3.8 Transferring Scan Results (Lines 143-149):

```
143 #The command below copy the scan results from the remote system to the local sytem and delete any previous files created.
144 echo 'Transferring scan and extracted results from remote system...'
145 sshpass -p "$REMPW" scp "$REMLOG@$REMIP":~/"$VIP-nmap" /home/kali
146 sshpass -p "$REMPW" scp "$REMLOG@$REMIP":~/"$VIP-whois" /home/kali
147 sshpass -p "$REMPW" ssh "$REMLOG@$REMIP" "rm $VIP*"
148 echo 'Transfer has been completed..'
149 echo ' '
```

- This section transfers the scan results (whois and nmap logs) from the remote server to the local system using sshpass and scp.
- It also deletes the temporary files created on the remote server after transferring them.

### 3.9 Completion and Log Location (Lines 151-164):

```
151 #The command below inform user that scanning is complete and provide file path of the logs.
152 #Creation of directory 'NRW' and its sub-directories for ease of collection of data withinn an unified folder.
153 echo -e "${BGRN}Scan complete.${CLR}"
154 DTSMP=$(date +%F-%H%M)
155 #LINE 157 can be commented out after initiation creation of the 'nrw folder' for subsequent execution
156 sudo mkdir /var/log/nrw
157 sudo mkdir /var/log/nrw/"$VIP-$DTSMP"
158 sudo mv ~/"$VIP"* /var/log/nrw/"$VIP-$DTSMP"
159 echo "The saved who-is and nmap logs has been saved and can be found here: "
160 WHO=$(find /var/log/nrw/"$VIP-$DTSMP" -name "$VIP"-whois)
161 NMP=$(find /var/log/nrw/"$VIP-$DTSMP" -name "$VIP"-nmap)
162 echo -e "${BCYN}$WHO${CLR}"
163 echo -e "${BCYN}$NMP${CLR}"
164 echo ' '
```

- This section informs the user that the scan is complete.
- It creates a timestamped directory (/var/log/nrw/$VIP-$DTSMP) to organize the scan results (whois and nmap logs).
- DTSMP=: This part assigns the output of the following command to a variable named DTSMP
    - $(date +%F-%H%M): This is a command substitution, meaning the output of the enclosed command will replace the entire expression.
    - date: This command retrieves the current date and time.

- +%F: This format specifier tells date to output the date in the format YYYY-MM-DD (ISO 8601 format).
- -%H%M: This part appends the hour and minute in 24-hour format (HHMM) to the date.

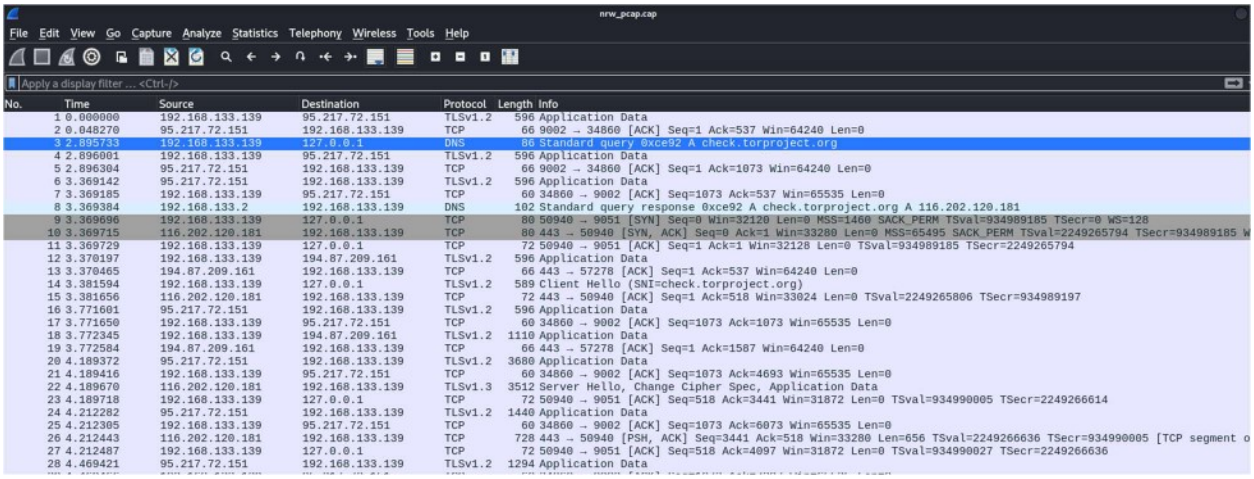
- It then moves the downloaded scan logs to the newly created directory.
- Finally, it displays the location paths of the saved whois and nmap logs for the user's reference.

### 3.10 Script Ending (Lines 165-168):

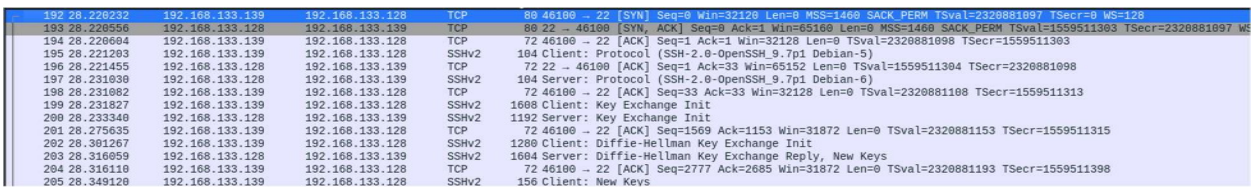
```
165 echo 'Exiting Script.'
166 echo 'End of Session. Have a nice day'
167
168 #End of Script
```

- This section informs the user that the script is exiting and provides a closing message.

## 4. Wireshark Analysis of Tcpdump (Pcap File) While Executing bash script: recon.sh



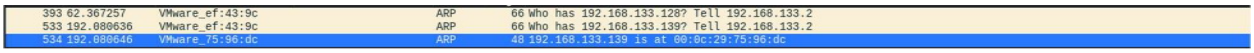

In the analysis, the local server (192.168.133.139) was observed to be sending a DNS query to find the corresponding IP address for torproject.org (116.202.120.181) that was executed between line 78 – 105 of the recon.sh script. This request travels through a hierarchy of DNS servers, starting locally and potentially reaching global servers if needed. Once the IP address is found, it's returned to the local server allowing the local server's IP address to remain anonymous.




During the use of the SSHPASS for SSH connection to the remote server under that is executed between line 131 – 141 of the recon.sh script. It was observed that a three-way handshake had

taken place to established, connect and complete the connection in the form of SYN, SYN-ACK, and ACK packets. First, the client (192.168.133.139) sends a SYN packet to initiate the connection. The server (192.168.133.128) responds with a SYN-ACK packet, acknowledging the client's request and proposing its own sequence number. Finally, the client sends an ACK packet to confirm the connection, and both devices can begin data transmission.

Upon further observation, it is interesting to note that during the ssh protocol connection (in this case SSHv2), Diffie-Hellman key init, exchange and reply were observed to have taken place. Based on further research, Diffie-Hellman in SSHv2 serves as the cornerstone for secure key exchange. It both parties to establish the shared secret key over an insecure channel (“Public Key Cryptography: Ensuring Secure Communication - Toktok.io”). The shared key will then then used to encrypt and decrypt data during the SSH session (*Key Agreement Purpose | Maßmöbel Stöckl*). By employing Diffie-Hellman, SSHv2 ensures that even if the session is intercepted, the eavesdropper cannot derive the secret key. This process is important in protecting the confidentiality and integrity of data transmitted over the network (“Overview of Propel’s GDPR and Enterprise Compliance Features”), safeguarding sensitive information from unauthorized access.




Lastly, Address Resolution Protocol (ARP) which acts as the translator between IP addresses and physical hardware addresses (MAC addresses) was observed to be intiated, When the local/remote server wants to send data to each other. Although the corresponding servers knows the IP address of the corresponding server. It needs the MAC address to physically locate the target device. ARP helps by broadcasting a request asking for the MAC address associated with the specific IP address. The device with the matching IP corresponds with the device’s MAC address (Vaněk), allowing the sender to transmit the data. This process ensures smooth communication within a local network.

## 5. Research of HTTP (Hypertext Transfer Protocol)

### 5.1 Fundamental Behavior of HTTP

HTTP (Hypertext Transfer Protocol) is designed to facilitate communication between web clients (like browsers) and web servers. Its primary purpose is to retrieve resources, such as HTML documents, images, and other media, from web servers (MDN Web Docs).

HTTP operates on a client-server model. A client, typically a web browser, initiates a request to a server, which hosts web resources. This request is sent as a text-based message, formatted according to HTTP specifications (“What Is Hypertext Transfer Protocol (HTTP)?”). The server processes the incoming request and generates a corresponding reply, also in text format.

A core principle of HTTP is its statelessness. This means that the server doesn't maintain any session information between requests. Each request is treated independently, without reference to previous interactions. While this simplifies server implementation, it also necessitates mechanisms like cookies and session management to track user information across multiple requests.

HTTP supports various request methods, each serving a specific purpose. The most common methods are GET, used to retrieve data, and POST, used to submit data to a server. Other methods include PUT, for updating data, DELETE, for removing data, and HEAD, for retrieving only the response headers (Cloudflare).

The protocol is flexible, allowing for custom headers to convey additional information. These headers can specify request details, response characteristics, or metadata about the transmitted data. HTTP also includes status codes in responses to indicate the outcome of the request, such as success, redirection, or error conditions.

Before HTTP, there was no standardized protocol for retrieving information from distributed systems. HTTP solved this by providing a universal method for accessing and transferring data over the internet.

The development and standardization of HTTP are documented in Request for Comments (RFC) documents. These RFCs outline the protocol specifications, including syntax, semantics, and message formats. The most widely used version is HTTP/1.1 (“Apache Tips”), defined in RFC 2616 (“Rfc2616”), which introduced features like persistent connections and pipelining to improve efficiency (“HTTP Specifications and Drafts”).

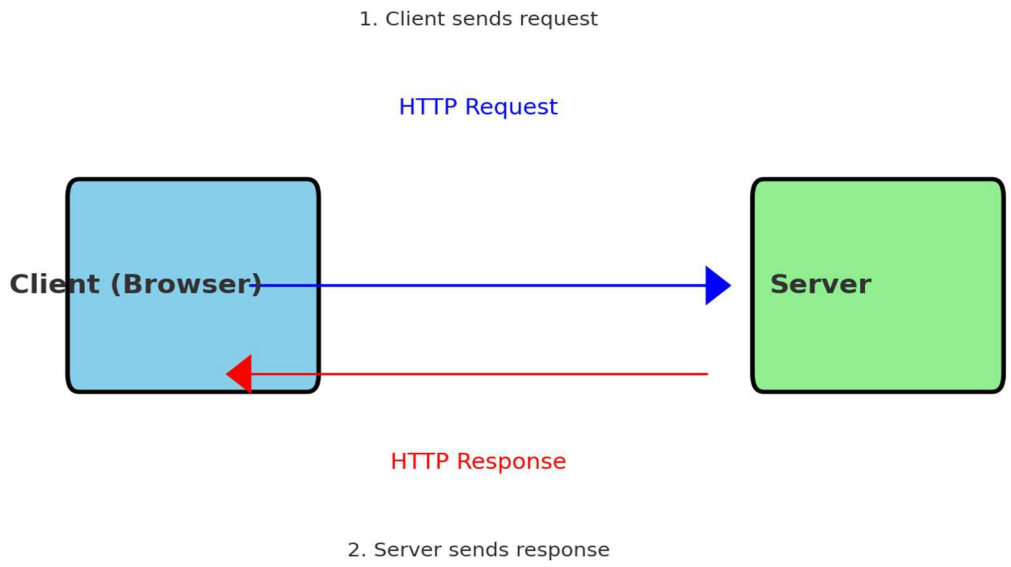
While HTTP has been instrumental in the growth of the web, its limitations, such as header overhead and connection management, have led to the development of newer protocols like HTTP/2 and HTTP/3. 1 (“HTTP Specifications and Drafts”).

In essence, HTTP provides a standardized framework for requesting and delivering web resources (“An Overview of HTTP”). Its simplicity, flexibility, and efficiency have contributed to its widespread adoption as the foundation of the World Wide Web.

### 5.2 HTTP Request and Response Cycle

Below are the basic steps involved in a typical HTTP request and response cycle. This cycle involves a client (usually a web browser) and a server. Here are the key components and steps we'll include:

1. **Client (Browser)**: Initiates the request.
2. **Server**: Receives the request and processes it.
3. **Request-Response Cycle**:
    - **Request**: The client sends an HTTP request to the server.
    - **Processing**: The server processes the request.
    - **Response**: The server sends back an HTTP response.




Here's a simple diagram illustrating the behavior of the HTTP protocol:

1. **Client (Browser)**: The client, often a web browser, initiates an HTTP request.
2. **HTTP Request**: The client sends an HTTP request to the server. This is depicted by the blue arrow.
3. **Server**: The server receives and processes the request.
4. **HTTP Response**: The server sends back an HTTP response to the client. This is depicted by the red arrow.

The labels provide a brief description of the actions in the request-response cycle. This diagram highlights the basic interaction between a client and a server using HTTP.

### 5.3 A Hypothetical HTTP Interaction Mechanism:

**Scenario:** A user wants to access the homepage of a fictional online bookstore, "BookWorm".

#### 5.3.1 Steps Involved:

1. **User Action:** The user types "www.bookworm.com [invalid URL removed]" into their web browser's address bar and presses Enter.
2. **DNS Resolution:** The browser queries the DNS to resolve "www.bookworm.com [invalid URL removed]" into its corresponding IP address.
3. **TCP Connection Establishment:** The browser initiates a TCP connection with the web server at the resolved IP address using the three-way handshake.

4. **HTTP Request:** The browser sends an HTTP GET request to the web server, specifying the desired resource (the homepage). The request might look like this:

```
GET / HTTP/1.1
Host: www.bookworm.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.0.0 Safari/537.36
```

5. **Server Processing:** The web server receives the request, processes it, and retrieves the homepage content.
6. **HTTP Response:** The server sends an HTTP response back to the browser. If successful, it will include a status code of 200 (OK) and the HTML content of the homepage. The response might look like this:

```
HTTP/1.1 200 OK
Content-Type: text/html

<!DOCTYPE html>
<html>
<head>
  <title>BookWorm</title>
</head>
<body>
  </body>
</html>
```

7. **Browser Rendering:** The browser receives the response, parses the HTML content, and displays the webpage to the user.
8. **Connection Closure:** After the page is loaded, the TCP connection is usually closed, although persistent connections can keep it open for subsequent requests.

This above is an example of the basic HTTP interaction. Real-world would involve more complex interactions, including redirects, cookies, and various HTTP methods.

## 5.4 Three Critical Flags or Options Affecting HTTP Behavior

### 5.4.1 Connection Header

The Connection header is crucial in determining the longevity of an HTTP connection. Traditionally, HTTP connections were short-lived, closing after a single request-response cycle. However, the Connection: keep-alive header introduced the concept of persistent connections, allowing multiple requests and responses to be sent over the same TCP connection, significantly improving performance (“Connection”). Conversely, Connection: close indicates that the connection should be terminated after the current transaction.

### 5.4.2 Cache-Control Header

The Cache-Control header empowers clients and servers to manage cached content efficiently. It includes directives like max-age, specifying the maximum age of a cached response, no-cache, preventing caching altogether, and public or private controlling ability to cache (“Cache-Control -

HTTP | MDN"). This header is instrumental in optimizing website performance and reducing server load by leveraging browser caching.

### 5.4.3 Content-Type Header

The Content-Type header is essential for correctly interpreting the content of an HTTP response ("Content-Type"). It specifies the media type of the entity-body, informing the client about the data format. Common content types include text/html, image/jpeg, application/json, and application/xml. Accurate content type identification is crucial for proper rendering and processing of the received data. Mismatched content types can lead to display errors or security vulnerabilities.

These three headers represent fundamental aspects of HTTP communication, influencing factors like performance, efficiency, and data handling. Their appropriate use is vital for optimizing web applications and ensuring correct content delivery.

## 5.5 HTTP Protocol: Strengths and Weaknesses in Relation to the CIA Triad

### 5.5.1 Strengths

1. **Availability:** HTTP is inherently designed for availability. It's a stateless protocol, meaning each request is independent, which enhances scalability and fault tolerance (Hashemi-Pour). Additionally, HTTP supports persistent connections, reducing connection overhead and improving response times, contributing to overall system availability.
2. **Integrity:** While HTTP itself doesn't provide end-to-end data integrity, it lays the groundwork for it. The protocol ensures that messages are delivered correctly and without modification between the client and server (Chauhan). This is achieved through mechanisms like sequence numbers and checksums within the TCP layer, which underlies HTTP.
3. **Partial Confidentiality:** While not providing strong encryption by default, HTTP can leverage HTTPS (HTTP Secure) to protect data confidentiality through encryption (Fortinet). This optional layer ensures that data is transmitted securely, protecting sensitive information from eavesdropping.

### 5.5.2 Weaknesses

1. **Confidentiality:** HTTP in its plain text form offers minimal confidentiality. Sensitive data transmitted over HTTP is susceptible to interception and disclosure ("Advantages of HTTP Protocol | Disadvantages of HTTP Protocol"). This is why HTTPS is essential for protecting sensitive information.
2. **Integrity:** While HTTP itself doesn't guarantee data integrity end-to-end, relying on lower-level protocols can introduce vulnerabilities. If these underlying protocols are compromised, data integrity can be affected. Additionally, HTTP doesn't provide mechanisms for detecting data tampering or replay attacks ("Preventing HTTPS Replay Attacks").

3. **Availability:** While HTTP is designed for availability, it can be susceptible to denial-of-service (DoS) attacks (Rozen). Overwhelming a server with requests can hinder its ability to respond to legitimate traffic, impacting availability.

It's important to note that while HTTP has inherent strengths and weaknesses, the overall security posture of an application or system depends on how HTTP is implemented and secured. Additional security measures, such as firewalls, intrusion detection systems, and application-level security, are essential to protect against threats.

## 6. Research of HTTPS (Hypertext Transfer Protocol Secure)

HTTPS stands for Hypertext Transfer Protocol Secure. It's the secure version of HTTP, the protocol used for transferring data between web browsers and websites.

The primary purpose of HTTPS is to enhance the security of data transmission. It achieves this by encrypting all data that passes between the browser and the website. This encryption process renders the data unreadable to anyone intercepting the communication, protecting sensitive information like passwords, credit card numbers, and personal messages Cloudflare (“What Is HTTPS?”).

By using HTTPS, websites can ensure that their users' data is transmitted securely using the cryptographic internet security protocol certificate, known as the Secure Socket Layer/Transport Layer Security (SSL/TLS) certificate which are secure digital identifiable object to builde trust and confidence (“What Is a SSL Certificate? SSL / TLS Certificate Explained - AWS”). This is especially crucial for websites that handle sensitive information, such as online banking, e-commerce platforms, and email services.

### 6.1 Demonstration of HTTPS Protocol Behaviour

In order to provide a step-by-step explanation of how a HTTPS protocol behaves, a written demonstration was created below to aid the readers understanding.

1. **Server Side**: A simple HTTPS server that uses a self-signed certificate for SSL/TLS encryption.
2. **Client Side**: A client that connects to the server over HTTPS and sends a request.

#### 6.1.1 Key Components

- **SSL/TLS Certificate**: We'll generate a self-signed certificate for demonstration purposes. In a production environment, a certificate from a trusted Certificate Authority (CA) would be used.
- **HTTPS Server**: Using Python's http.server and ssl libraries.
- **HTTPS Client**: Using Python's requests library with SSL verification.

#### 6.1.2 Using of Self-Signed Certificates

For simple explanation purposes, self-signed certificates will be used to demonstrate the connection locally. This example can be expanded for more complex applications, including proper certificate management.

### 6.1.3 Implementation

Let's start by implementing the server and client.

## 6.2 HTTPS Server Implementation

Here's the Python code written for a simple HTTPS server application:

```
import http.server
import ssl

# Define the handler to serve HTTP requests
class SimpleHTTPRequestHandler(http.server.SimpleHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.end_headers()
        self.wfile.write(b"Hello, HTTPS world!")

# Create the server
server_address = ('localhost', 4443)
httpd = http.server.HTTPServer(server_address, SimpleHTTPRequestHandler)

# Generate or provide your SSL certificate and key
# For demonstration, we're using self-signed certificates
httpd.socket = ssl.wrap_socket(httpd.socket,
                               keyfile="path/to/server-key.pem",
                               certfile='path/to/server-cert.pem',
                               server_side=True)

print("Serving on https://localhost:4443")
httpd.serve_forever()
```

### 6.2.1 Importing Required Modules

- **http.server:** This module provides basic classes for creating HTTP servers. It includes classes for handling HTTP requests and responses.
- **ssl:** This module provides access to Transport Layer Security (previously known as Secure Sockets Layer, SSL) encryption and secure communication.

### 6.2.2. Defining the Request Handler

- **SimpleHTTPRequestHandler:** This class inherits from http.server.SimpleHTTPRequestHandler, which handles HTTP GET requests.
- **do_GET(self):** This method is called when an HTTP GET request is received. It overrides the parent class's method to provide a custom response.
    - **self.send_response(200):** Sends an HTTP status code 200 (OK) to the client, indicating the request was successful.

- **self.send_header("Content-type", "text/html"):** Specifies the content type of the response, which in this case is HTML.
- **self.end_headers():** Signals the end of the headers section and sends them to the client.
- **self.wfile.write(b"Hello, HTTPS world!"):** Sends the response body. The b before the string indicates that it is a byte string, which is required when writing data to 'wfile'.

### 6.2.3. Creating the Server

- **server_address:** A tuple specifying the server's address and port. Here, it is set to run on localhost (the local machine) at port 4443.
- **httpd:** An instance of http.server.HTTPServer that binds the server to the specified address and associates it with the SimpleHTTPRequestHandler class to handle incoming requests.

### 6.2.4. Enabling SSL/TLS Encryption

- **ssl.wrap_socket**: This function wraps the server's socket, enabling SSL/TLS encryption. It takes several parameters:
  - **httpd.socket**: The original server socket to be wrapped.
  - **keyfile**: The path to the private key file (server-key.pem). This key is used to decrypt data sent by the client.
  - **certfile**: The path to the certificate file (server-cert.pem). This certificate is sent to the client to authenticate the server.
  - **server_side=True**: Indicates that this socket will be used for a server-side SSL/TLS connection.

### 6.2.5. Starting the Server

- **print(...):** Outputs a message indicating that the server is running and listening for connections at the specified address and port.
- **httpd.serve_forever():** Starts the server and keeps it running, processing incoming requests. It will continue to run until it is manually stopped.

### 6.3 HTTPS Client Implementation

Here's the Python code for a simple HTTPS client:

```python
import requests

# URL of the server (change if needed)
url = "https://localhost:4443"

# Send a GET request to the server
response = requests.get(url, verify='path/to/server-cert.pem')

# Print the response from the server
print(f"Response status code: {response.status_code}")
print(f"Response content: {response.text}")
```

#### 6.3.1 Importing the requests Library

- requests: A popular Python library for making HTTP/HTTPS requests. It abstracts the complexities of making requests behind a simple API, allowing you to send HTTP requests and handle responses easily.

#### 6.3.2 Setting the URL of the Server

- url: A string that contains the URL of the server to which the GET request will be sent. In this case, the URL is https://localhost:4443, indicating that the server is running on the local machine (localhost) and is listening on port 4443. The use of https:// indicates that the connection should be secured using SSL/TLS.

#### 6.3.3 Sending a GET Request

- **requests.get(url, verify='path/to/server-cert.pem')**: This function sends an HTTP GET request to the specified URL.
  - **url**: The URL to which the GET request is sent.
  - **verify**: This parameter is used to specify the path to the server's SSL/TLS certificate. It ensures that the client's connection to the server is secure and that the server's identity is verified.
    - If the verify parameter is set to False, the SSL certificate verification will be disabled, which is **not recommended** for production environments as it can lead to security vulnerabilities.
    - In this example, the verify parameter is set to 'path/to/server-cert.pem', which should be replaced with the actual path to the certificate file used by the server.

### 6.3.4. Handling the Response

- **response**: The variable that holds the response object returned by the requests.get function.
    - **response.status_code**: The HTTP status code returned by the server. It indicates the result of the HTTP request (e.g., 200 for a successful request, 404 for not found).
    - **response.text**: The content of the response, decoded to a string. This is the body of the response, typically HTML, JSON, or other data formats.

The code then prints the status code and the content of the response received from the server.

## 6.4 Steps to Run the Application

### 6.4.1 Generate Self-Signed Certificate

Use OpenSSL or another tool to generate a self-signed certificate and private key. For example:

```
openssl req -new -x509 -keyout server-key.pem -out server-cert.pem -days 365 -nodes
```

### 6.4.2 Command Breakdown

**openssl**: This is the command-line tool for using the OpenSSL library, which implements the SSL and TLS protocols. It includes a variety of cryptographic operations, including the creation and management of private keys, public keys, certificates, and more.

**req**: This specifies that you want to use the openssl req command, which is primarily used for creating and processing certificate signing requests (CSRs). In this case, it's being used to generate a self-signed certificate.

**-new**: This flag tells OpenSSL to generate a new certificate request.

**-x509**: This option is used to indicate that the output should be a self-signed certificate instead of a certificate signing request. X.509 is a standard format for public key certificates, which includes the public key and the identity associated with it.

**-keyout server-key.pem**: This specifies the output file for the generated private key. In this case, the private key will be saved in a file named server-key.pem. The private key is an essential part of the SSL/TLS encryption process, used to decrypt data encrypted with the public key.

**-out server-cert.pem**: This specifies the output file for the generated self-signed certificate. The certificate, along with the public key, is saved in a file named server-cert.pem. This certificate is what the server will provide to clients to establish a secure connection.

**-days 365**: This sets the validity period of the certificate to 365 days. After this period, the certificate will expire and need to be renewed or replaced.

**-nodes**: This stands for "no DES" and indicates that the private key should not be encrypted. Normally, private keys can be encrypted with a passphrase for added security, but using this option means the generated key will not be encrypted. This makes it easier to use the key without needing to enter a passphrase but is less secure because if the key file is compromised, it can be used directly.

### 6.4.3 Mode of Execution

**Start the HTTPS Server**: Run the server code. Ensure that the certificate and key paths are correctly specified.

**Run the HTTPS Client**: Execute the client code. Make sure the verify parameter points to the correct certificate file for SSL verification.

## 6.5 Basic Demonstration of HTTPS Behavior

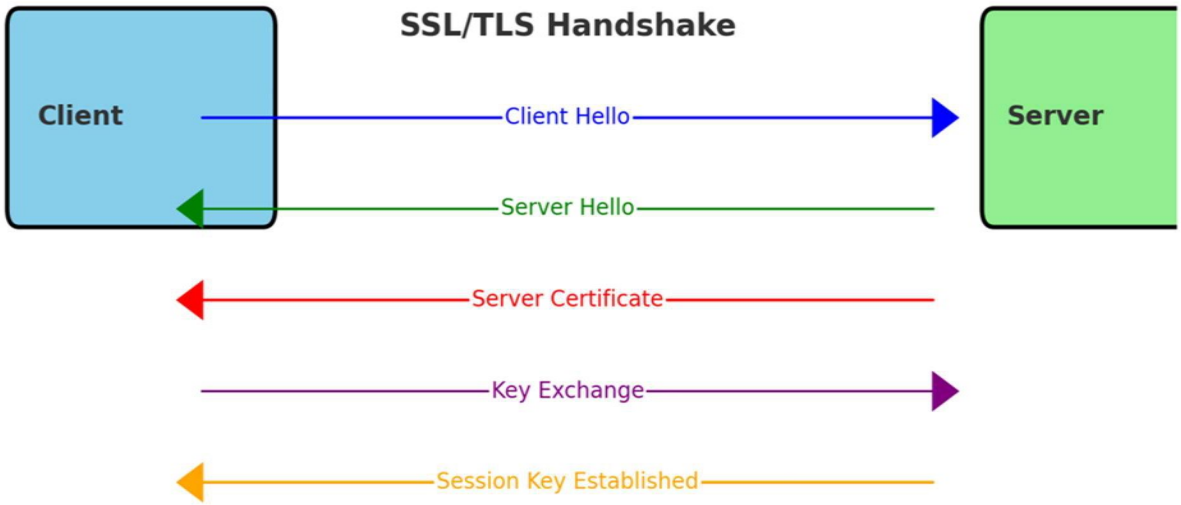
The basic demonstrated will be represented in the two diagrams that will be displayed below in the form of a SSL/TLS handshake diagram and a data exchange diagram:

### 6.5.1 Components Involved

The components involved are as follows:

- **Encrypted Communication**: The connection between the client and server is encrypted using SSL/TLS, ensuring data privacy and integrity.
- **SSL/TLS Handshake**: Before exchanging data, the client and server perform an SSL/TLS handshake to establish a secure connection.
- **Certificate Verification**: The client verifies the server's certificate to ensure it is communicating with the correct server.

### 6.5.2 SSL/TLS Handshake Diagram:



The above diagram demonstrates the key steps in the SSL/TLS handshake, including:

- Client Hello
- Server Hello
- Server Certificate
- Key Exchange
- Session Key Establishment

### 6.5.3 Data Exchange Diagram:


The diagram above illustrates the secure data exchange after the handshake, including:

- **Client Request (Encrypted)**: After establishing a secure connection, the client sends an encrypted request to the server.
- **Server Response (Encrypted)**: The server responds with encrypted data.

These diagrams illustrate the secure communication process in HTTPS, ensuring that data transmitted between the client and server is encrypted and secure.

## 7. HTTPS' Impact Resolution to HTTP Weakness (in relation to CIA Triad)

HTTPS significantly enhances the security posture of HTTP by addressing its confidentiality and integrity shortcomings. By employing encryption, HTTPS safeguards data from interception and disclosure, ensuring confidentiality (Cloudflare, "Why Is HTTP Not Secure? | HTTP vs. HTTPS"). This encryption process involves converting data into an unreadable format, rendering it useless to unauthorized parties.

Additionally, HTTPS incorporates mechanisms to verify the authenticity of the communicating parties, protecting against man-in-the-middle attacks (Imperva). Regarding integrity, HTTPS provides data integrity through cryptographic hashes and message authentication codes. These techniques detect any modifications to the transmitted data, ensuring its authenticity and reliability. By combining encryption and data integrity measures, HTTPS establishes a secure communication channel, mitigating the vulnerabilities inherent in the plaintext HTTP protocol.

## 8. Recommendations

Prioritize HTTPS Implementation over HTTP for any web application. This is not just a recommendation but a necessity in today's digital landscape where data security and privacy are paramount. HTTPS not only protects data integrity and confidentiality but also improves search engine optimisation rankings and boosts user trust.

Use Valid SSL/TLS Certificates ensure that your HTTPS implementation uses a valid SSL/TLS certificate from a trusted Certificate Authority (a.k.a CA) (SSL Support Team). Regularly update and renew these certificates to avoid lapses that could lead to security warnings for users.

Implement and use HSTS (HTTP Strict Transport Security) to enforce HTTPS usage by informing browsers to only connect to your site using HTTPS. This prevents users from being tricked into visiting unsecured versions of your website.

Prepare for Advanced Security Needs: While the provided code examples offer a foundational understanding of HTTPS implementation, real-world applications often require more complex security measures. Be prepared to implement additional layers of security, such as multi-factor authentication, intrusion detection systems, and data encryption at rest.

## 9. Conclusion

Understanding the fundamental differences between HTTP and HTTPS is crucial for building secure and reliable web applications. While HTTP provides the foundation for web communication, its lack of encryption makes it vulnerable to attacks. HTTPS, on the other hand, addresses these shortcomings by establishing a secure, encrypted connection between the client and server. This is paramount for protecting sensitive user data, such as financial information, personal details, and login credentials.

By implementing HTTPS, developers can significantly enhance the security and trustworthiness of their websites, fostering user confidence and compliance with industry regulations. The provided code examples offer a foundational understanding of HTTPS implementation, but real-world applications often require more complex security measures and best practices.