

计网：HTTP 服务器实验报告

一：实验要求

1.1 小组人员及分工：

1.2 题目

实现：使用 C 语言实现最简单的 HTTP 服务器

1. 同时支持 HTTP（80 端口）和 HTTPS（443 端口）

使用两个线程分别监听各自端口

2. 只需支持 GET 方法，解析请求报文，返回相应应答及内容

3. 支持的状态码

需支持的状态码	场景
200 OK	对于443端口接收的请求，如果程序所在文件夹存在所请求的文件，返回该状态码，以及所请求的文件
301 Moved Permanently	对于80端口接收的请求，返回该状态码，在应答中使用Location字段表达相应的https URL
206 Partial Content	对于443端口接收的请求，如果所请求的为部分内容（请求中有Range字段），返回该状态码，以及相应的部分内容
404 Not Found	对于443端口接收的请求，如果程序所在文件夹没有所请求的文件，返回该状态码

1.3 实验流程

1. 根据上述要求，实现 HTTP 服务器程序

2. 执行 `sudo python topo.py` 命令，生成包括两个端节点的网络拓扑

3. 在主机 h1 上运行 HTTP 服务器程序，同时监听 80 和 443 端口 h1

`# ./http-server`

4. 在主机 h2 上运行测试程序，验证程序正确性 h2 `# python3 test/test.py` 如果没有出现 `AssertionError` 或其他错误，则说明程序实现正确

5. 在主机 h1 上运行 `http-server`，所在目录下有一个小视频（30 秒左右）

6. 在主机 h2 上运行 `vlc`（注意切换成普通用户），通过网络获取并播放该小视频媒体 -> 打开网络串流 -> 网络 -> 请输入网络 URL -> 播放

7. 抓包，并对抓到的包进行解密

二：实验环境安装

2.1 VMWare 和 Ubuntu20.04 安装

1. 下载 Ubuntu 镜像

<https://mirrors.tuna.tsinghua.edu.cn/ubuntu-releases/>

2. 安装虚拟机/配置

3. 安装 Ubuntu 进 VMWare 软件中

2.2 mininet 安装

```
sudo apt update
sudo apt install python3-pip          //并没有预装 pip
sudo apt install vim
sudo apt install git
sudo apt install vlc
sudo apt install xorg    //下载 x11
```

用 git 安装 mininet:

mininet2.3.0

https://blog.csdn.net/m0_52479012/article/details/115521881

git clone <https://github.com/mininet/mininet> 下载

cd mininet

git tag # 列举出当前的 mininet 版本

git checkout -b mininet-2.3.0 2.3.0 # 选择 2.3.0 版本进行下载

cd ..

<https://www.cnblogs.com/xrszff/p/11258975.html>

为了使得 mininet 支持 python3

修改 mininet/util/install.sh 中的 PYTHON=\${PYTHON:-python3}

#####把 url 全部的 git 协议改成 https 协议

git config --global url."https://".insteadOf git://

./mininet/util/install.sh -a //进行完整安装

sudo mn --test pingall //测试是否成功

三：设计思路

3.1 SOCKET 基础知识

为什么要用套接字 socket?

因为套接字提供了一种在网络上进行通信的机制,提供了存储有关套接字连接的所有必要信息的方式。

Socket 是应用层与 TCP/IP 协议族通信的中间软件抽象层,是一种编程接口。Socket 屏蔽了不同网络协议的差异支持面向连接(TCP)和无连接(UDP)。

3.2 HTTP 基础知识

最简单的 HTTP 服务器至少应该能够监听 HTTP 请求（通常是 TCP 端口 80 或 443），接收客户端（如浏览器）的请求，解析这些请求，并根据请求发送相应的响应。这通常包括提供静态文件（如 HTML 文件、图片等）和生成 HTTP 响应头。使用两个线程分别监听各自端口只需支持 GET 方法，解析请求报文，返回相应应答及内容。客户端（如浏览器）通过发送一个 GET 请求来向服务器请求资源，资源有网页、视频、图片。GET 请求是请求行：包括方法（GET），请求的资源的 URL，以及 HTTP 协议版本。例如：GET /index.html HTTP/1.1+请求头：包含关于请求的附加信息，如客户端能接受的内容类型、浏览器类型等。请求头以空行结束。例如：

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;
q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

HTTP 服务器的响应包含：

1. 状态行（Status Line）：包含了协议版本（如 HTTP/1.1），一个状态码（如 200, 404 等），以及一个状态消息（如 OK, Not Found 等）。
2. 响应头（Headers）：一系列的键值对，提供关于响应的额外信息，例如 Content-Type（内容类型），Content-Length（内容长度），Server（服务器信息），Date（日期和时间）等。
3. 空行（Empty Line）：分隔响应头和响应体的一个空行。
4. 响应体（Body）：实际返回的数据。对于网页请求，通常是 HTML 文本。在您的例子中，如果 index.html 被请求，响应体将包含该 HTML 文件的内容。

3.3 HTTP 重定向 HTTPS

从实验要求中可以看出有必要实现 **http 到 https 的自动重定向**。

你（作为客户端）要去一个朋友（服务器）家（网站）玩。你按照朋友之前告诉你的地址（http://10.0.0.1/index.html，即使用 HTTP 协议的地址）去找他。当你到达那个地址时，你发现门上贴着一张纸条（服务器的重定向响应），上面写着：“我搬家了，请到我的新地址找我：https://10.0.0.1/index.html”（即 HTTPS 协议的地址）。服务器告诉你，你应该使用更安全的方式（HTTPS）去访问同一资源的新位置。通常情况下，如果你是按照常规程序（默认设置）行事，你会自动按照纸条上的新地址去朋友的新家。如果我们通过设置 allow_redirects=False 告诉你（客户端）不要自动按照纸条上的指示去新地址，而是先停下来，把纸条（重定向信息）拿回来。通过这个过程，你可以确保你的朋友真的搬家了，并且他的新家（新网站）是用一个更安全的方式（HTTPS）访

问的。

如果在 `requests.get` 方法中没有设置 `allow_redirects=False`, 那么当服务器响应一个重定向请求（例如，从 HTTP 到 HTTPS 的 301 永久重定向）时，`requests` 库会自动跟随这个重定向。这意味着，如果服务器返回一个指向 HTTPS 版本的 Location 头，`requests` 客户端会自动尝试访问这个新的 HTTPS URL。

3.4 SSL 证书

如何理解 ssl 证书？

你正在网上购物或进行网上银行交易。这就像是你要通过一条街道（互联网）去一个商店（网站）购物。在这条街道上，可能有一些不怀好意的人（黑客）试图偷听你和商店之间的对话，比如你的信用卡信息。为了确保你和商店之间的对话是安全的，商店会在门口展示一份“安全许可证”（SSL 证书）。这个证书有点像是一个官方身份证明，它告诉你这家商店是合法和可信的，不是一个假冒的商店。这个证书由一个可信的第三方机构（证书颁发机构）发出，这个机构的角色就像是一个公认的身份证明机关。当客户端与服务器建立连接时，它会检查这个证书。这个过程就是 SSL 证书验证。如果证书有效、没有过期，并且是由一个可信的证书颁发机构签发的，那么你的浏览器就会认为这个连接是安全的。这时，你和商店之间的所有信息交换都会被加密，这意味着即使有人偷听你们的对话，他们也无法理解这些信息。但如果证书有问题（比如过期了，或者不是由一个可信的机构签发的），浏览器会警告你这个连接可能不安全。

3.5 公钥与私钥

`cnlab.cert` 公钥是证书的一部分，客户端（如浏览器）连接到服务器时，服务器会提供这个证书作为身份的证明。客户端会验证这个证书的有效性，确保正在与正确的服务器通信，并使用证书中的公钥来加密信息

`cnlab.prikey` 私钥在服务器上保密存储，不应该被泄露给其他任何人。它用于解密客户端发送到服务器的加密信息，以及在使用数字签名时验证服务器的身份

在我们的项目中，从后面的报文抓包中我们可以发现，报文加密使用的是临时的私钥而不是 `cnlab.prikey`。

四、项目内容与执行方法

4.1 项目内容

项目内容中除了老师提供的文件外，还包括：

`http_server.h` 服务器头文件

`http_server.c` 服务器源代码文件

`playvideo.html` 我做的播放视频嵌入 video 的网页

`video.mp4` 我上传的视频文件

`encode.pcapng` 在一次实验用 wireshark 抓到的包文件

`sslkeylog.log` 在该次实验中在 vlc 所在客户端导出的日志文件，包含该次回话的临时私钥内容，可用来解析包文件

./test/mytest.py 我自己写的客户端 python 测试文件，用于实现我自己的目的

4.2 实验流程

1. 安装 gcc: `sudo apt install build-essential`, 用 gcc 编译 http_server 代码 `gcc -g http_server.c -o http_server -lssl -lcrypto`
2. 生成 http_server 可执行文件
3. 生成包含两个端节点虚拟主机 h1 h2 的网络拓扑 `sudo python3 topo.py`
4. 打开 mininet 主机 h1 和 h2 的终端窗口, h1 是服务器, h2 是客户端 `xterm h1`
`xterm h2`
(mininet> h1 ip addr 可以看出 h1 默认的 ip 地址就是 10.0.0.1/8)
5. 在 h1 的终端窗口下用 `sudo ./http_server` 命令来启动 http 服务器
6. 在 h2 的终端窗口下用 `python3 test/test.py` 进行测试
7. `python3 test/mytest.py` 进行我自己的测试
8. 在主机 h2 上 `su - xx` 切换到 ubuntu 的非 root 用户
9. 在 h2 中 `export DISPLAY=:0`, 可以使得当前终端能够使用 X11 转发图形界面
10. 在 Ubuntu 机器上打开一个新的终端窗口, 运行 `xhost` 命令: 在终端中输入以下命令并执行: `xhost +`, 这个命令会禁用 X11 服务器的访问控制, 允许所有用户 (包括远程用户和虚拟主机上的用户) 连接到您的 X11 会话
11. 在 h2 中 `vlc -vvv` 打开 vlc 图形化界面并在终端打印详细日志
12. 在 vlc 图形界面中, 点击 media-open network stream, 输入 <http://10.0.0.1/video.mp4> 即可打开视频
13. 再开一个 h2 窗口, 使用 wireshark 抓包监听 h2 的网卡, 选择 443 端口进行监听

4.3 实验代码

HTTP 代码:

```
void http_server()
{
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_addr_len;
    int listenfd, connectfd;
    char buf[MAXLINE], first_line[MAXLINE], left_line[MAXLINE],
method[MAXLINE], url[MAXLINE], version[MAXLINE];
    long bytesRead;
    int reuseAddrOption = 1;
    pid_t pid;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);
    setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &reuseAddrOption,
sizeof(reuseAddrOption)); // 设置 socket 选项以允许地址重用

    bzero(&server_addr, sizeof(server_addr)); // 结构体清零
    // 对 servaddr 结构体进行赋值
```

```

server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
server_addr.sin_port = htons(SERV_PORT1);

Bind(listenfd, (struct sockaddr *)&server_addr,
sizeof(server_addr)); // 将 listenfd 绑定到 server_addr 指定的地址和
端口
Listen(listenfd, BACKLOGSIZE);
// 开始监听来自客户端的连接请求

// 死循环中进行 accept()
while (1)
{
    client_addr_len = sizeof(client_addr);

    // accept() 函数返回一个 connectfd 文件描述符
    connectfd = Accept(listenfd, (struct sockaddr *)&client_addr,
&client_addr_len);
    setsockopt(connectfd, SOL_SOCKET, SO_REUSEADDR,
&reuseAddrOption, sizeof(reuseAddrOption));

    pid = fork();
    if (pid < 0)
    {
        printf("fork is error");
    }
    else if (pid == 0)
    { // pid=0 表示子进程
        while (1)
        {
            bytesRead = Read(connectfd, buf, MAXLINE); // 从连接套
接字读取数据
            if (bytesRead == 0) // 如果没有
数据, 表示客户端已关闭连接
            {
                printf("client closed.\n");
                break;
            }
            sscanf(buf, "%s %s %s", method, url, version); // 解析
HTTP 请求方法、URL 和版本
            printf("method = %s\n", method); // 打印
请求方法
            printf("url = %s\n", url); // 打印
请求 URL

```

```

        printf("version = %s\n", version);           // 打印
HTTP 版本
                                                    // 检
查是否是 HTTP GET 请求
        if (strcasecmp(method, "GET") == 0 && strstr(version,
"HTTP"))
        {
            struct sockaddr_in addr;
            socklen_t addr_size = sizeof(struct sockaddr_in);
            getsockname(connectfd, (struct sockaddr *)&addr,
&addr_size);

            char server_ip[INET_ADDRSTRLEN];
            inet_ntop(AF_INET, &addr.sin_addr, server_ip,
INET_ADDRSTRLEN);
            // 现在 server_ip 包含了服务器的 IP 地址
            printf("host ip is = %s\n", server_ip);
            if (strcasecmp(method, "GET") == 0 && strstr(version,
"HTTP"))
            {
                // 构建并发送 301 Moved Permanently 响应
                char response[2048]; // 因为 url 长度设置也是
MAXLINE, response 要比 MAXLINE 大
                snprintf(response, sizeof(response),
                    "HTTP/1.1 301 Moved Permanently\r\n"
                    "Location: https://%s%s\r\n\r\n",
                    server_ip, url);
                printf("Response headers:\n%s", response);
                send(connectfd, response, strlen(response),
0);

                Close(connectfd); // 发送完直接关闭 因为 HTTP
协议是无连接的

                break;
            }
        }
        // Close(connectfd);
    }

    else
    { // pid>0 表示父进程
        Close(connectfd);
    }
}

```

```
}
```

HTTPS 代码:

```
void https_server()
{
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_addr_len;
    int listenfd, connectfd;
    char buf[MAXLINE], first_line[MAXLINE], left_line[MAXLINE],
method[MAXLINE], url[MAXLINE], version[MAXLINE];
    long bytesRead;
    pid_t pid;
    int reuseAddrOption = 1;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);
    bzero(&server_addr, sizeof(server_addr)); // 结构体清零
    // 对 servaddr 结构体进行赋值
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port = htons(SERV_PORT2);

    Bind(listenfd, (struct sockaddr *)&server_addr,
sizeof(server_addr));
    Listen(listenfd, BACKLOGSIZE);

    // 死循环中 accept()
    while (1)
    {
        client_addr_len = sizeof(client_addr);

        // accept() 函数返回一个 connectfd 文件描述符
        connectfd = Accept(listenfd, (struct sockaddr *)&client_addr,
&client_addr_len);
        if (connectfd < 0)
        {
            perror("Error in accept");
            exit(1);
        }

        setsockopt(connectfd, SOL_SOCKET, SO_REUSEADDR,
&reuseAddrOption, sizeof(reuseAddrOption));
        // 初始化 SSL 会话
        SSL *ssl = load_SSL_certification(connectfd);
        if (!ssl)
        {
```



```

        perror("SSL initialization failed");
        Close(connectfd); // 关闭 socket
        exit(1);
    }

    pid = fork();
    if (pid < 0)
    {
        printf("fork is error");
    }
    else if (pid == 0)
    { // pid=0 表示子进程
        while (1)
        {
            // 进行 SSL 握手，确保 SSL 连接安全建立
            if (SSL_accept(ssl) == -1)
            {
                ERR_print_errors_fp(stderr);
            }

            // 从 SSL 连接中读取数据
            bytesRead = SSL_Read(ssl, buf, MAXLINE);
            if (bytesRead == 0)
            {
                printf("the client close\r\n\r\n");
                break;
            }
            // printf("%s\n", buf);
            sscanf(buf, "%s %s %s", method, url, version);
            printf("method = %s\n", method);
            printf("url = %s\n", url);
            printf("version = %s\n", version);

            if (strcasecmp(method, "GET") == 0)
            {
                https_response(ssl, buf, url); // 响应客户端
            }
        }
        close(connectfd);
        SSL_free(ssl);
    }

    else
    { // pid>0 表示是父进程

```

```

        close(connectfd);
        SSL_free(ssl);
    }
}
}

```

五：项目测试

5.1 利用 python 模拟客户端测试

在虚拟主机 h1 中 `sudo ./http_server` 启动服务器后，在 h2 的终端窗口下用 `python3 test/test.py` 进行测试，没有出现 `AssertionError` 或其他错误，程序实现正确。

并且我编写了文件 `mytest.py` 添加了对于视频类型 GET 请求的测试，代码如下：

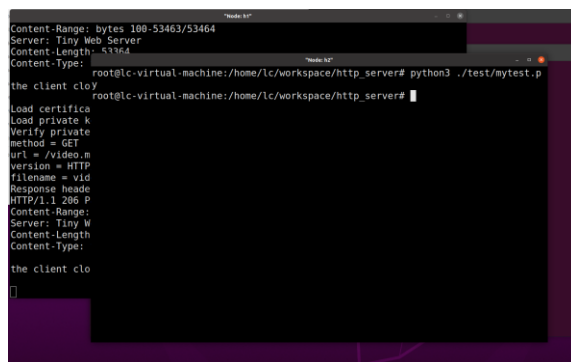
```

# http video
headers = { 'Range': 'bytes=0-' }
r = requests.get('https://10.0.0.1/video.mp4', headers=headers)
assert(r.status_code == 206 and open(test_dir + '/../video.mp4',
'rb').read()[0:] == r.content)

```

在 http 服务器源代码编写中，我加入了打印报文头的功能，可以在 h1 终端中看到重要的打印信息，在进行 `mytest.py` 的测试中，打印的内容如下所示：

h2 终端：



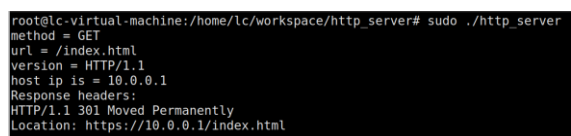
```

Content-Range: bytes 108-53463/53464
Server: Tiny Web Server
Content-Length: 53354
Content-Type:
the client clo
root@lc-virtual-machine:/home/lc/workspace/http_server# python3 ./test/mytest.p
root@lc-virtual-machine:/home/lc/workspace/http_server#
Load certifica
Load private k
Verify private
method = GET
url = /video.m
version = HTTP
filename = vid
Response heade
HTTP/1.1 206 P
Content-Range:
Server: Tiny W
Content-Length
Content-Type:
the client clo

```

可见正常退出，没有出现 `AssertionError` 或其他错误

h1 终端：对于 http301 请求



```

root@lc-virtual-machine:/home/lc/workspace/http_server# sudo ./http_server
method = GET
url = /index.html
version = HTTP/1.1
host ip is = 10.0.0.1
Response headers:
HTTP/1.1 301 Moved Permanently
Location: https://10.0.0.1/index.html

```

回复 301，并且 URL 的 location 也

是对的，而且客户端没有继续跟踪 https 新 url

对于 http200 请求

```

Content-Type: text/html
the client close
method = GET
url = /index.html
version = HTTP/1.1
host ip is = 10.0.0.1
Response headers:
HTTP/1.1 301 Moved Permanently
Location: https://10.0.0.1/index.html

Load certification
Load private key
Verify private key
method = GET
url = /index.html
version = HTTP/1.1
filename = index.html
Response headers:
HTTP/1.1 200 OK
Server: Tiny Web Server
Content-Length: 53464
Content-Type: text/html

```

先返回 301，然后跟踪新 url 返回 200，是符合标准的, 并且响应头和响应体的内容也是正确的

对于 http404 请求

```

method = GET
url = /notfound.html
version = HTTP/1.1
host ip is = 10.0.0.1
Response headers:
HTTP/1.1 301 Moved Permanently
Location: https://10.0.0.1/notfound.html

Load certification
Load private key
Verify private key
method = GET
url = /notfound.html
version = HTTP/1.1
filename = notfound.html
can not find file

method = GET
url = /dir/index.html
version = HTTP/1.1
host ip is = 10.0.0.1
Response headers:

```

先返回 301，跟踪新 url 后返回 404，是正确的

访问 dir 文件夹内的 html 文件

```

method = GET
url = /dir/index.html
version = HTTP/1.1
host ip is = 10.0.0.1
Response headers:
HTTP/1.1 301 Moved Permanently
Location: https://10.0.0.1/dir/index.html

Load certification
Load private key
Verify private key
method = GET
url = /dir/index.html
version = HTTP/1.1
filename = dir/index.html
Response headers:
HTTP/1.1 200 OK
Server: Tiny Web Server
Content-Length: 53464
Content-Type: text/html

```

由输出可知也是正确的

测试我写的 http 视频代码

```

Load certification
Load private key
Verify private key
method = GET
url = /video.mp4
version = HTTP/1.1
filename = video.mp4
Response headers:
HTTP/1.1 206 Partial Content
Content-Range: bytes 0-4287572/4287573
Server: Tiny Web Server
Content-Length: 4287573
Content-Type: video/mp4

the client close

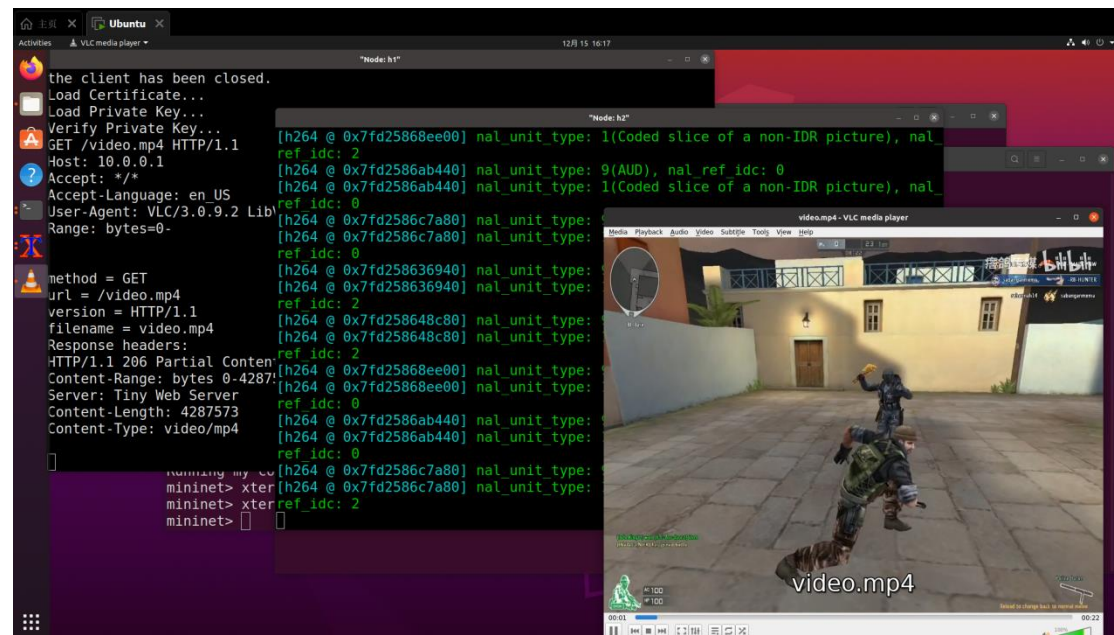
```

可知响应体 range 值 0-4287572 是

正确的，而且状态码也是 206，通过了 python 测试，可见对于视频类的 GET 请求，测试也是通过的。

5.2 vlc 播放视频

在虚拟主机 h2 中用 vlc 成功打开视频文件，视频流畅播放，而且 http 服务器终端打印的内容也符合要求。



5.3 浏览器播放视频

打开火狐浏览器，浏览器地址栏输入 <https://10.0.0.1/video.mp4>，网站会提示证书不安全，然后我们需要手动信任证书。并进行一些设置：

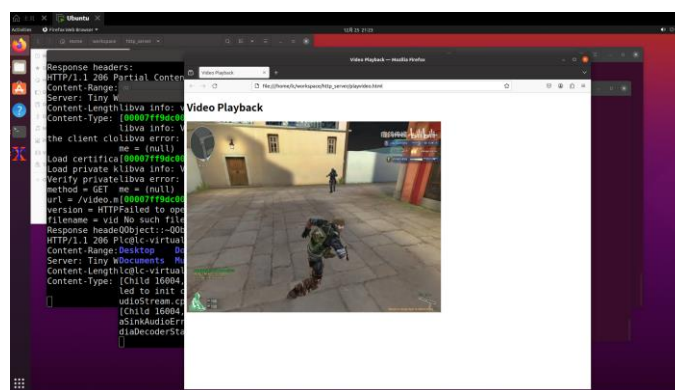
打开新标签页，输入 `about:config` 并接受风险警告。

搜索 `security.ssl.enable_ocsp_stapling`，将其设置为 `false`。

搜索 `security.cert_pinning.enforcement_level`，将其设置为 `0`。

搜索 `network.stricttransportsecurity.preloadlist`，将其设置为 `false`。

之后就可以正常播放视频，我还编写了 `playvideo.html` 文件将视频嵌入到网站中，`firefox ./playvideo.html`，使用这条命令调用火狐浏览器，就可以看到网页打开视频正常播放，效果图如下：



六：报文分析

在 h2 中启动 wireshark，监听 h2 的网卡 h2-eth0，选择了 443 端口，抓到了报文，报文分析如下：

第 1 条报文是客户端发的,可以看出使用 tcp 协议,类型为 ipv4,目的端口 443,标志位中只有 SYN 被置位,因此这是 TCP SYN 包,用于建立到端口 443 的连接。这是 TCP 三次握手过程中的第一步

[illegible]

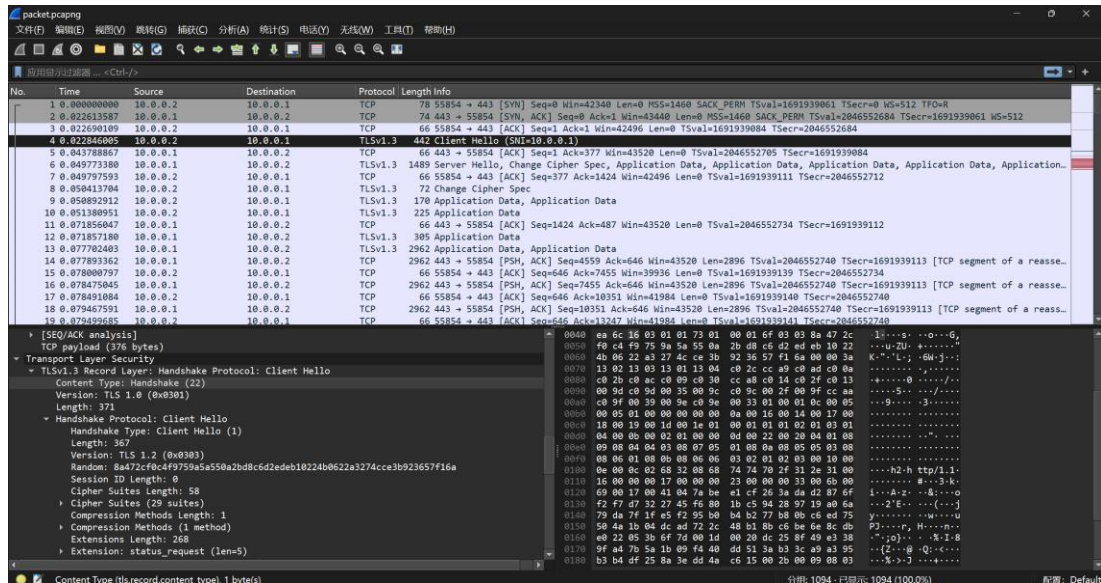
```

Flags: 0x002 (SYN)
 000. .... = Reserved: Not set
...0 .... = Accurate ECN: Not set
.... 0... = Congestion Window Reduced: Not set
.... .0.. = ECN-Echo: Not set
.... ..0. = Urgent: Not set
.... ...0 = Acknowledgment: Not set
.... ....0.. = Push: Not set
.... .....0... = Reset: Not set
▶ .... ....1. = Syn: Set
.... .....0 = Fin: Not set

```

第 2 条报文是服务器（10.0.0.1）对客户端（10.0.0.2）的 SYN-ACK
第 3 条报文是从 10.0.0.2 发送到 10.0.0.1 的 TCP 报文，用于确认（ACK）从 10.0.0.1 接收到的 SYN-ACK。这是 TCP 三次握手过程中的最后一步，完成了连接的建立

第 4 条报文是 TLS 握手过程中的第一步，“Client Hello”消息是客户端发送到服务器的第一个消息，用于开始握手，还包含客户端支持的 TLS 版本、随机数等



TLS 握手是通过刚刚建立的 TCP 连接通信的，TLS 握手的目的是安全地验证通信双方的身份，协商加密算法，以及生成用于加密通信的密钥

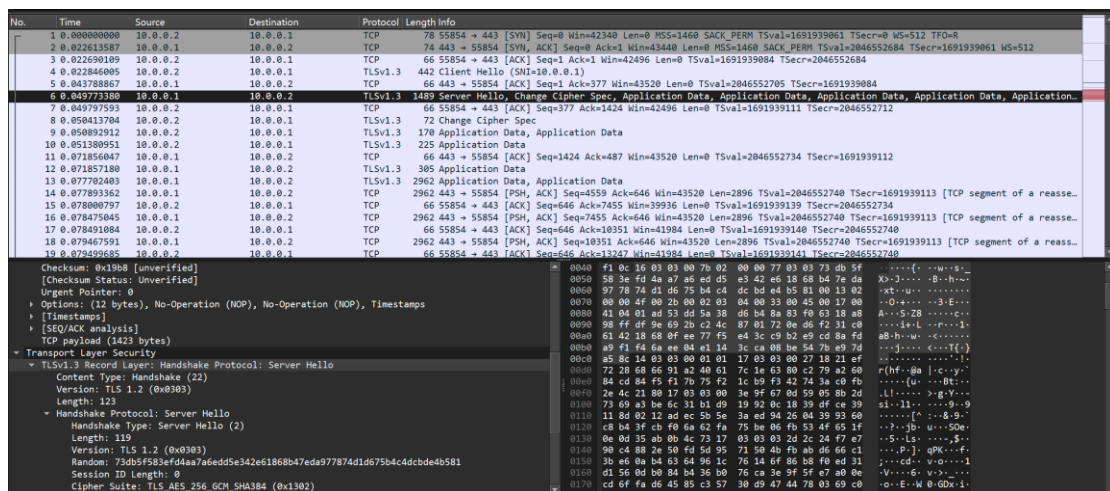
第 6 条报文是 TLS 握手的第二步，服务器确认了 TLS 版本，提供了一个随机数，选择了一个密码套件，并可能发送了一些 TLS 扩展信息，通过 Server Hello，服务器与客户端开始建立一个加密的通信会话。

这则报文还有 Change Cipher Spec，它是 TLS 握手过程的一部分，告知接收方，发送方准备切换到新协商的安全参数进行加密通信，从此时起，接下来的记录将被加密

Certificate：服务器提供其证书链，允许客户端验证服务器的身份

Certificate Verify：服务器通过此消息证明它拥有其证书中的公钥对应的私钥。这是对服务器身份的额外验证。

Finished：这个消息包含一个加密的哈希值，用于验证握手过程的完整性。它标志着服务器端握手过程的结束。



获得 tls 临时密钥，解密 https 报文

之后我发现了我的服务器配置使用了支持前向保密的 TLS 密码套件，就算我有服务器的私钥(cnlab.prikey)，也无法解密 TLS 流量。这是因为前向保密密码套件

会为每个会话生成唯一的临时密钥，而不是直接使用服务器的长期密钥。
我的解决办法是设置环境变量（如 SSLKEYLOGFILE），然后将该变量指向一个文件路径。启动 vlc 播放视频后，进行 TLS 通信时产生的所有密钥都会被写入这个文件(我命名为 sslkeylog.log)。就这样我获得了 TLS 通信时产生的所有密钥。

在客户端虚拟主机 h2 设置环境变量
export SSLKEYLOGFILE=/home/lc/workspace/http_server/sslkeylog.log
在 Wireshark 中，通过 Preferences -> Protocols -> TLS，然后在
“(Pre)-Master-Secret log filename” 字段中指定包含 TLS 密钥的日志文件路径，即 sslkeylog.log，就完成了对报文的解密。
解密后的报文可以看到 http 明文，如图所示：

1	0.000000000	10.0.0.2	10.0.0.1	TCP	78	34384 -> 443 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM TSval=1696936560 TSecr=0 WS=512 TFO=R
2	0.021485446	10.0.0.1	10.0.0.2	TCP	74	443 -> 34384 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 SACK_PERM TSval=2051550182 TSecr=1696936560 WS=512
3	0.021567557	10.0.0.2	10.0.0.1	TCP	66	34384 -> 443 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=1696936582 TSecr=2051550182
4	0.021804893	10.0.0.2	10.0.0.1	TLSv1.3	442	Client Hello (SNI=10.0.0.1)
5	0.042748303	10.0.0.1	10.0.0.2	TCP	66	443 -> 34384 [ACK] Seq=1 Ack=377 Win=43520 Len=0 TSval=2051550203 TSecr=1696936582
6	0.049553835	10.0.0.1	10.0.0.2	TLSv1.3	1489	Server Hello, Change Cipher Spec, Encrypted Extensions, Certificate Request, Certificate, Certificate Verify
7	0.049596452	10.0.0.2	10.0.0.1	TCP	66	34384 -> 443 [ACK] Seq=377 Ack=1424 Win=42496 Len=0 TSval=1696936610 TSecr=2051550210
8	0.050204483	10.0.0.2	10.0.0.1	TLSv1.3	72	Change Cipher Spec
9	0.050871076	10.0.0.2	10.0.0.1	TLSv1.3	170	Certificate, Finished
10	0.051629705	10.0.0.2	10.0.0.1	HTTP	225	GET /video.mp4 HTTP/1.1
11	0.071141229	10.0.0.1	10.0.0.2	TCP	66	443 -> 34384 [ACK] Seq=1424 Ack=487 Win=43520 Len=0 TSval=2051550233 TSecr=1696936611
12	0.07193189	10.0.0.1	10.0.0.2	TLSv1.3	305	New Session Ticket
13	0.076874432	10.0.0.1	10.0.0.2	TCP	2962	[TLS segment of a reassembled PDU]
14	0.07696658	10.0.0.2	10.0.0.1	TCP	66	34384 -> 443 [ACK] Seq=646 Ack=4559 Win=41984 Len=0 TSval=1696936637 TSecr=2051550233
15	0.077097498	10.0.0.1	10.0.0.2	TCP	2962	443 -> 34384 [PSH, ACK] Seq=4559 Ack=646 Win=43520 Len=2896 TSval=2051550238 TSecr=1696936612 [TCP segment of a
16	0.077239453	10.0.0.1	10.0.0.2	TCP	2962	443 -> 34384 [PSH, ACK] Seq=7455 Ack=646 Win=43520 Len=2896 TSval=2051550238 TSecr=1696936612 [TCP segment of a
17	0.077480722	10.0.0.1	10.0.0.2	TCP	2962	443 -> 34384 [PSH, ACK] Seq=10351 Ack=646 Win=43520 Len=2896 TSval=2051550238 TSecr=1696936612 [TCP segment of a
18	0.077561921	10.0.0.2	10.0.0.1	TCP	66	34384 -> 443 [ACK] Seq=646 Ack=13247 Win=35840 Len=0 TSval=1696936638 TSecr=2051550238

10	0.051629705	10.0.0.2	10.0.0.1	HTTP	225	GET /video.mp4 HTTP/1.1
99	0.184818792	10.0.0.2	10.0.0.1	HTTP	231	GET /video.mp4 HTTP/1.1
118	0.271401444	10.0.0.1	10.0.0.2	HTTP	474	HTTP/1.1 206 Partial Content
120	0.272392629	10.0.0.2	10.0.0.1	HTTP	229	GET /video.mp4 HTTP/1.1
1655	1.337819375	10.0.0.1	10.0.0.2	HTTP	8387	HTTP/1.1 206 Partial Content (video/mp4)
1657	21.038468626	fe80::3806:a7ff:fea::ff02::12		ICMPv6	70	Router Solicitation from 3a:06:a7:a6:d2:47
1	0.000000000	10.0.0.2	10.0.0.1	TCP	78	34384 -> 443 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM TSval=1696936560 TSecr=0 WS=512 TFO=R
2	0.021485446	10.0.0.1	10.0.0.2	TCP	74	443 -> 34384 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 SACK_PERM TSval=2051550182 TSecr=1696936560 WS=512
3	0.021567557	10.0.0.2	10.0.0.1	TCP	66	34384 -> 443 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=1696936582 TSecr=2051550182

在加密的报文中，这些都被封装在 Application Data

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.2	10.0.0.1	TCP	78	34384 -> 443 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM TSval=1696936560 TSecr=0 WS=512 TFO=R
2	0.021485446	10.0.0.1	10.0.0.2	TCP	74	443 -> 34384 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 SACK_PERM TSval=2051550182 TSecr=1696936560 WS=512
3	0.021567557	10.0.0.2	10.0.0.1	TCP	66	34384 -> 443 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=1696936582 TSecr=2051550182
4	0.021804893	10.0.0.2	10.0.0.1	TLSv1.3	442	Client Hello (SNI=10.0.0.1)
5	0.042748303	10.0.0.1	10.0.0.2	TCP	66	443 -> 34384 [ACK] Seq=1 Ack=377 Win=43520 Len=0 TSval=2051550203 TSecr=1696936582
6	0.049553835	10.0.0.1	10.0.0.2	TLSv1.3	1489	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data, Appli
7	0.049596452	10.0.0.2	10.0.0.1	TCP	66	34384 -> 443 [ACK] Seq=377 Ack=1424 Win=42496 Len=0 TSval=1696936610 TSecr=2051550210
8	0.050204483	10.0.0.2	10.0.0.1	TLSv1.3	72	Change Cipher Spec
9	0.050871076	10.0.0.2	10.0.0.1	TLSv1.3	170	Application Data, Application Data
10	0.051629705	10.0.0.2	10.0.0.1	TLSv1.3	225	Application Data
11	0.071141229	10.0.0.1	10.0.0.2	TCP	66	443 -> 34384 [ACK] Seq=1424 Ack=487 Win=43520 Len=0 TSval=2051550233 TSecr=1696936611
12	0.07193189	10.0.0.1	10.0.0.2	TLSv1.3	305	Application Data
13	0.076874432	10.0.0.1	10.0.0.2	TCP	2962	Application Data, Application Data
14	0.07696658	10.0.0.2	10.0.0.1	TCP	66	34384 -> 443 [ACK] Seq=646 Ack=4559 Win=41984 Len=0 TSval=1696936637 TSecr=2051550233
15	0.077097498	10.0.0.1	10.0.0.2	TCP	2962	443 -> 34384 [PSH, ACK] Seq=4559 Ack=646 Win=43520 Len=2896 TSval=2051550238 TSecr=1696936612 [TCP segment of a
16	0.077239453	10.0.0.1	10.0.0.2	TCP	2962	443 -> 34384 [PSH, ACK] Seq=7455 Ack=646 Win=43520 Len=2896 TSval=2051550238 TSecr=1696936612 [TCP segment of a
17	0.077480722	10.0.0.1	10.0.0.2	TCP	2962	443 -> 34384 [PSH, ACK] Seq=10351 Ack=646 Win=43520 Len=2896 TSval=2051550238 TSecr=1696936612 [TCP segment of a
18	0.077561921	10.0.0.2	10.0.0.1	TCP	66	34384 -> 443 [ACK] Seq=646 Ack=13247 Win=35840 Len=0 TSval=1696936638 TSecr=2051550238

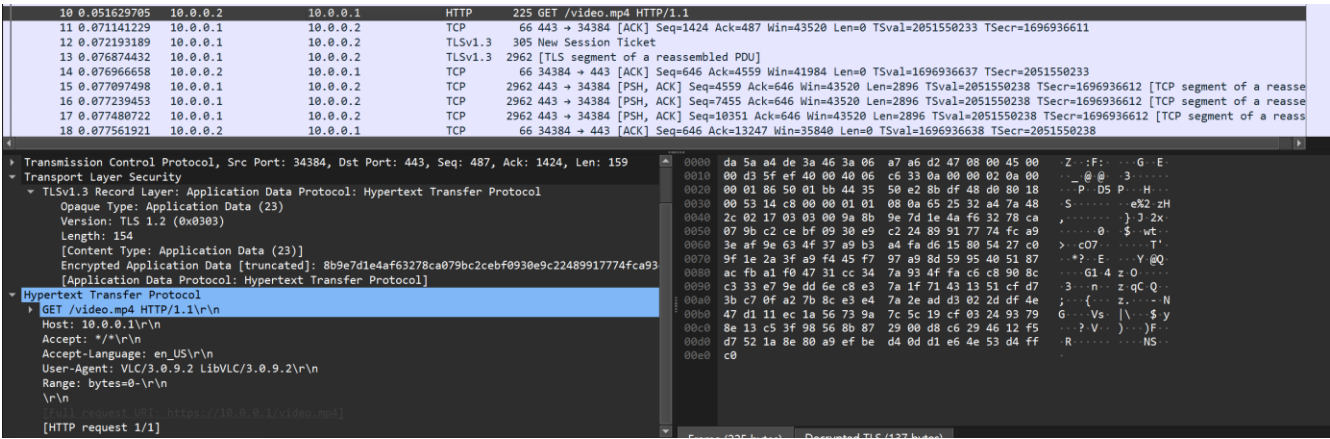
第 9 条报文包含 TLS Certificate, Finished, Certificate” 消息：客户端在双向 TLS 认证中发送其证书给服务器
“Finished” 消息是 TLS 握手的最后一个步骤，用来确认之前的握手消息没有被篡改，并且双方都已准备好进行加密通信。
至此 finished 消息服务器和客户端都发送过了，TLS 连接建立彻底完成。

总结 TLS 握手的步骤：
1 客户端开始 TLS 握手，发送 “Client Hello” 消息。这个消息包括客户端支持的 TLS 版本，提议的加密套件，以及可能的扩展，如服务器名称指示（SNI）。
2 服务器响应 “Client Hello” 消息，并发送一系列消息，包括 “Server Hello”， “Change Cipher Spec”， “Encrypted Extensions”， “Certificate Request”， “Certificate”， “Certificate Verify”， 以及 “Finished”
这些消息共同完成了服务器端的握手过程，表明服务器已经验证了自己的身份（通过证书和证书验证），也要求客户端验证其身份，并准备好开始加密通信。
3 客户端完成握手，客户端发送 “Certificate” 消息和 “Finished” 消息。这表示客户端已经验证了服务器的身份，也提供了自己的身份验证信息，并确认了

握手过程。

完成这些步骤后，TLS 握手就完成了，客户端和服务端都验证了对方的身份（进行了双向认证），并且协商了一个共享的密钥用于之后的加密通信，接下来的通信将使用 TLS 协议加密，确保数据的安全性和完整性。

第 10 条报文是解密出来的 http 明文，是在 TLS 握手完成后，客户端（VLC 媒体播放器）向服务器发送的第一个 HTTP GET 请求，http 请求头部分可以看到服务器地址是 10.0.0.1，UA 表明发送请求的客户端是 VLC 媒体播放器，版本号为 3.0.9.2，Range: bytes=0-：这意味着请求的是从第 0 字节开始的文件内容，一般用于视频流媒体。能看到 url 是 https://10.0.0.1/video.mp4



第 12 条报文是服务器发给客户端的 New Session Ticket，告诉客户端一个新的会话票据，客户端可以使用这个票据来在未来快速恢复会话，而无需再次执行完整的 TLS 握手过程。

接下来是很多 TCP 协议的报文，进行视频流的传输，数据传输通常伴随着 TCP 的[PSH, ACK]标志，表示“推送”数据到接收方，并期望一个 ACK 响应，这是流媒体视频播放时的数据传输模式。

Seq 和 Ack 是 TCP 协议的序列号和确认号，用于确保数据传输的可靠性。报文中反复出现的[TLS segment of a reassembled PDU]表明数据量较大，需要多个 TCP 包来传输。Win 值的变化反映了接收方动态调整其接收缓冲区大小，以适应网络条件和数据处理速度。

服务器以分段的形式发送数据，vlc 确认接收每个数据段，这个过程保证了数据的连续传输和播放的流畅性。视频数据作为响应体通过 TLS 加密的连接传输，VLC 客户端会接收这些数据并进行播放。

在这个过程中，TLS 层确保数据的安全性和完整性，而 HTTP 层则负责按照 Web 标准传输数据。VLC 作为客户端，在接收到数据后负责解码和播放视频。

可以看到 http 的服务器响应报文明文，如图


```
10.0.0.1 10.0.0.2 10.0.0.1 HTTP 225 GET /video.mp4 HTTP/1.1
10.0.0.1 10.0.0.2 10.0.0.1 HTTP 231 GET /video.mp4 HTTP/1.1
10.0.0.1 10.0.0.2 10.0.0.2 HTTP 474 HTTP/1.1 206 Partial Content
10.0.0.1 10.0.0.2 10.0.0.1 HTTP 229 GET /video.mp4 HTTP/1.1
10.0.0.1 10.0.0.2 10.0.0.2 HTTP 8387 HTTP/1.1 206 Partial Content (video/mp4)
1657 21.038468626 fe80::3806:a7ff:fea::f02:12 ICHPv6 70 Router Solicitation from 3a:06:a7:a6:d2:47
1 0.000000000 10.0.0.2 10.0.0.1 TCP 78 34384 -> 443 [SYN] Seq=0 Win=42496 Len=0 MSS=1460 SACK_PERM TSval=1696936568 TSecr=0 WS=512 TFO=0
2 0.021485446 10.0.0.1 10.0.0.2 TCP 74 443 -> 34384 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 SACK_PERM TSval=2051550182 TSecr=1696936568 WS=512
3 0.021567557 10.0.0.2 10.0.0.1 TCP 66 34384 -> 443 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=1696936582 TSecr=2051550182
5 0.042748383 10.0.0.1 10.0.0.2 TCP 66 443 -> 34384 [ACK] Seq=1 Ack=377 Win=43520 Len=0 TSval=2051550203 TSecr=1696936582
7 0.049596452 10.0.0.2 10.0.0.1 TCP 66 34384 -> 443 [ACK] Seq=377 Ack=1424 Win=42496 Len=0 TSval=1696936510 TSecr=2051550210
11 0.071141229 10.0.0.1 10.0.0.2 TCP 66 443 -> 34384 [ACK] Seq=1424 Ack=487 Win=43520 Len=0 TSval=2051550233 TSecr=1696936611
14 0.076966658 10.0.0.2 10.0.0.1 TCP 66 34384 -> 443 [ACK] Seq=646 Ack=4559 Win=41984 Len=0 TSval=1696936637 TSecr=2051550233
15 0.077897498 10.0.0.1 10.0.0.2 TCP 2962 443 -> 34384 [PSH, ACK] Seq=4559 Ack=646 Win=43520 Len=2896 TSval=2051550238 TSecr=1696936612 [TCP segment of a reassembled]
16 0.077239453 10.0.0.2 10.0.0.1 TCP 2962 443 -> 34384 [PSH, ACK] Seq=7455 Ack=646 Win=43520 Len=2896 TSval=2051550238 TSecr=1696936612 [TCP segment of a reassembled]
17 0.077480722 10.0.0.1 10.0.0.2 TCP 2962 443 -> 34384 [PSH, ACK] Seq=10351 Ack=646 Win=43520 Len=2896 TSval=2051550238 TSecr=1696936612 [TCP segment of a reassembled]
18 0.077561921 10.0.0.2 10.0.0.1 TCP 66 34384 -> 443 [ACK] Seq=646 Ack=13247 Win=35840 Len=0 TSval=1696936638 TSecr=2051550238
19 0.077969543 10.0.0.1 10.0.0.2 TCP 1514 443 -> 34384 [ACK] Seq=13247 Ack=646 Win=43520 Len=1448 TSval=2051550238 TSecr=1696936612 [TCP segment of a reassembled]

Response Version: HTTP/1.1
Status Code: 206
[Status Code Description: Partial Content]
Response Phrase: Partial Content
Content-Range: bytes 79914-4287572/4287573\r\n
Server: Tiny Web Server\r\n
Content-Length: 4207659\r\n
[Content Length: 4207659]
Content-Type: video/mp4\r\n
\r\n
[HTTP response 2/2]
[Time since request: 1.265426746 seconds]

[Request URI: https://10.0.0.1/video.mp4]
File Data: 4207659 bytes
[Expert Info (Note/Malformed): HTTP body subdissector failed, trying heuristic subdissector]
Media Type
```

可以看到响应头是 206 Partial Content，允许 VLC 获取视频文件的特定部分，而不是整个文件，这对于实现视频的快速加载和缓冲非常重要，特别是在观看者寻找视频的特定部分时。在流媒体播放中，客户端不需要下载整个视频文件就开始播放；相反，它会下载视频的一部分，然后逐渐下载更多内容，同时播放已下载的部分。

后面还有 tls alert 报文。close_notify 警告是一种特殊的 TLS 警告，用于指示发送方即将关闭这个 TLS 连接。这是一种关闭 TLS 连接的方式，确保双方都知道通信即将结束。

```
1658 25.888499892 10.0.0.2 10.0.0.1 TLSv1.3 90 Alert (Level: Warning, Description: Close Notify)
1659 25.888703607 10.0.0.2 10.0.0.1 TCP 66 34388 -> 443 [FIN, ACK] Seq=839 Ack=4215557 Win=4664320 Len=0 TSval=1696962449 TSecr=2051551689
1660 25.9099986212 10.0.0.1 10.0.0.2 TCP 66 443 -> 34388 [FIN, ACK] Seq=4215557 Ack=840 Win=43520 Len=0 TSval=2051576070 TSecr=1696962449
1661 25.909942268 10.0.0.2 10.0.0.1 TCP 66 34388 -> 443 [ACK] Seq=840 Ack=4215558 Win=4664320 Len=0 TSval=1696962470 TSecr=2051576070

Frame 1658: 90 bytes on wire (720 bits), 90 bytes captured (720 bits) on interface h2-eth0, id 0
Ethernet II, Src: 3a:06:a7:a6:d2:47 (3a:06:a7:a6:d2:47), Dst: da:5a:a4:de:3a:46 (da:5a:a4:de:3a:46)
Internet Protocol Version 4, Src: 10.0.0.2, Dst: 10.0.0.1
Transmission Control Protocol, Src Port: 34388, Dst Port: 443, Seq: 815, Ack: 4215557, Len: 24
Transport Layer Security
  TLSv1.3 Record Layer: Alert (Level: Warning, Description: Close Notify)
    Opaque Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
    Length: 19
    [Content Type: Alert (21)]
    Alert Message
      Level: Warning (1)
      Description: Close Notify (0)
```

七：遇到的问题及解决方案

mininet 下载问题：

默认用 apt 下载 mininet 有一些问题，解决这些问题花了一些时间，主要原因是因为不兼容 git 协议引起的，后来我使用 git clone <https://github.com/mininet/mininet> 选择合适的分支，才成功下载了 mininet。gdb 的时候显示端口已经被占用：

我使用 gdb 工具帮我分析调试 c 代码，如果 GDB 设置为在 fork 时跟踪父进程或者在 fork 时完全脱离子进程，则子进程会继续独立于 GDB 运行。即使退出 GDB，这些子进程仍然会继续执行。

GDB 终止时未杀死子进程：当退出 GDB 时，它可能没有发送终止信号给所有由它启动或跟踪的进程。这可能导致这些进程继续在后台运行。只能手动杀死与端口 80 和 443 有关的进程。

不了解 socket 编程

经过一系列的学习，我了解了 socket 对于网络连接方面提供的各种方法，并运用到编程中去

不了解 response 中报文头部的构造

之前因为 response 响应头构造不规范，导致客户端总是收不到合格的响应，后来查阅资料了解了响应头的规范构造，学会了使用 sprintf 去拼接响应头，最终正确实现 http 的响应。

vlc 图形化界面问题

刚开始使用 vlc 的时候遇到了很多问题，比如没有 X11 无法打开图形化界面，而且就算安装了，在虚拟主机 h2 中也不能调用 X11 打开 vlc 界面，之后我查阅资料，通过 xhost +和在 h2 中定义环境变量的方法成功地调用了 X11，打开了 vlc 的图形界面。

八：总结

在这次实验中，我们团队共同构建了一个在 mininet 上的 HTTP 服务器，从而深入理解了网络编程和 HTTP 通信协议的基本原理。学习了如何使用套接字进行网络通信，包括创建、绑定、监听和接受连接。在处理 HTTP 请求和响应方面，掌握了构建服务器的关键步骤。也遇到了一些挑战，但最终完美解决。