

基于OpenSSL的安全Web服务器程序技术报告

基于OpenSSL的安全Web服务器程序技术报告

一、源码分析

文件架构

- WebServer根目录

- 代码文件

- OpenSSL 使用到的文件

代码分析

- Common.h

 - 完整代码

 - HTTPS 相关定义

- HttpProtocol.h

 - 完整代码

- HttpProtocol.cpp

二、可调整的地方

- 重复定义

- 无用内容

- 文件不存在直接结束服务器

- 请求路径中存在 .. 直接结束服务器

- Content-Type类型无法显示bug

三、WebServer的测试

- 正常GET请求——默认路径 /WebServer

 - Web捕获到的请求信息

 - 返回的信息

- 请求其他文件——favicon.ico

 - Web捕获到的请求信息

 - 返回额信息

 - 类型缺失bug

四、报文分析

- 一次完整请求的报文交互过程

 - TCP三次握手，四次挥手

 - 完整的SSL过传输过程

五、后续工作

POST功能完善

- 添加对应的定义

- 添加对应判断

- 修改bool CHttpProtocol::SSLRecvRequest逻辑

- 服务端展示效果

- 客户端请求

Content-Type无法显示Bug修复

- 问题硕源

- 解决方法

- 效果展示

入口函数冗余分析

- 原因

非法请求

- 请求文件不存在

- 请求文件的路径中存在 ..

总结

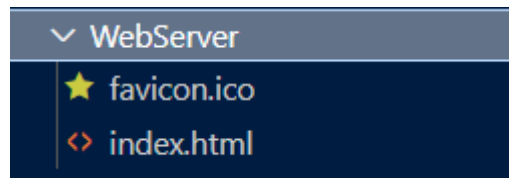
一、源码分析

文件架构

WebServer根目录

HTTP请求的所有资源都在这个目录下，测试时需要测试不同的请求资源，则将对应的文件放在此目录下。

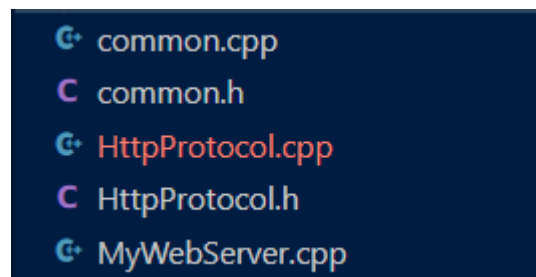
该目录下初始仅有一张 logo 图片， 和一个 html 文件



代码文件

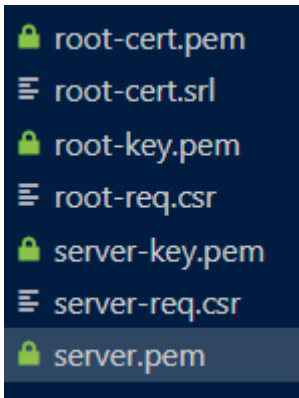
主要涉及编写的代码文件如下：

- Common.cpp
- Common.h : 定义了一系列类型以及常量
- HttpProtocol.h: 定义了一个HTTP协议类
- HttpProtocol.cpp: 主要的功能实现
- MyWebServer.cpp: 入口文件，开启整个Web
-



OpenSSL 使用到的文件

- root-cert.pem : 根证书
- root-key.pem : 根密钥
- server.pem : 服务器证书
- server-key.pem: 服务器密钥



代码分析

Common.h

完整代码

```
1 #ifndef _common_h
2 #define _common_h
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <errno.h>
7 #include <sys/types.h>
8 // #include <sys/socket.h>    // 套接字
9 #include <winsock2.h>    //注释掉的头文件windows系统下都在winsock2.h中
10 // #include <netinet/in.h>
11 // #include <netinet/tcp.h> // tcp
12 // #include <netdb.h>
13 #include <fcntl.h>
14 #include <signal.h>
15 #include <unistd.h>
16 #include <string.h>
17 #include <pthread.h>
18
19 // #include <arpa/inet.h>
20
21 #include <openssl/ssl.h>
22 #include <openssl/bio.h>
23 #include <openssl/err.h>
24
25 #define HTTPS_PORT 8000
26 #define METHOD_GET 0
27 #define METHOD_HEAD 1
28
29 #define BUFSIZE 1024
30 #define ROOTCERTPEM "root-cert.pem"
31 #define ROOTKEYPEM "root-key.pem"
32 #define SERVERKEYPEM "server-key.pem"
33 #define SERVERPEM "server.pem"
34 #define PASSWORD "OPENSSL"
35
36 #define HTTP_STATUS_OK "200 OK"
37 #define HTTP_STATUS_CREATED "201 Created"
38 #define HTTP_STATUS_ACCEPTED "202 Accepted"
```

```

39 #define HTTP_STATUS_NOCONTENT "204 No Content"
40 #define HTTP_STATUS_MOVEDPERM "301 Moved Permanently"
41 #define HTTP_STATUS_MOVEDTEMP "302 Moved Temporarily"
42 #define HTTP_STATUS_NOTMODIFIED "304 Not Modified"
43 #define HTTP_STATUS_BADREQUEST "400 Bad Request"
44 #define HTTP_STATUS_UNAUTHORIZED "401 Unauthorized"
45 #define HTTP_STATUS_FORBIDDEN "403 Forbidden"
46 #define HTTP_STATUS_NOTFOUND "404 File can not found!"
47 #define HTTP_STATUS_SERVERERROR "500 Internal Server Error"
48 #define HTTP_STATUS_NOTIMPLEMENTED "501 Not Implemented"
49 #define HTTP_STATUS_BADGATEWAY "502 Bad Gateway"
50 #define HTTP_STATUS_UNAVAILABLE "503 Service Unavailable"
51
52 typedef int INT;
53 typedef unsigned int UINT;
54 typedef unsigned int *PUINT; // 无符号int的指针
55 typedef unsigned long DWORD; // 双字 4字节
56 typedef unsigned int UINT; // 无符号int
57 typedef UINT SOCKET; // 套接字， 无符号int
58 typedef unsigned long DWORD; // 双字
59 typedef int BOOL; // 用int表示bool
60 typedef unsigned char BYTE; // char 表示一个字节
61 typedef unsigned short WORD; // short 表示一个字
62 typedef float FLOAT; // float
63 typedef FLOAT *PFLOAT; // float的指针
64 typedef BOOL *PBOOL; // bool的指针，本质是int的指针
65 typedef BOOL *LPBOOL; // 可能是为了和windowsApi的命名习惯相符
66 typedef BYTE *PBYTE; // 字节指针
67 typedef BYTE *LPBYTE;
68 typedef int *PINT; // int 指针
69 typedef int *LPINT;
70 typedef WORD *PWORD;
71 typedef WORD *LPWORD;
72 typedef long *LPLONG;
73 typedef DWORD *PDWORD;
74 typedef DWORD *LPDWORD;
75 typedef void *LPVOID;
76 typedef char *LPSTR; // 字符串
77 typedef struct sockaddr *LPSOCKADDR; // 套接字地址信息
78 typedef void *HANDLE; // 句柄
79 #define INVALID_HANDLE_VALUE (HANDLE) - 1
80 #define INVALID_FILE_SIZE (DWORD)0xFFFFFFFF
81 #define INVALID_SOCKET (SOCKET)(~0)
82 #define SOCKET_ERROR (-1)
83 typedef struct REQUEST
84 {
85     HANDLE hExit;
86     SOCKET Socket; // 存储socket连接
87     int nMethod; // 表示请求所使用的方法， 对应get和head， 后续添
    加的post等应该还需要额外添加状态
88     DWORD dwRecv; // 已经接收的字节数
89     DWORD dwSend; // 已经发送的字节数
90     int hFile; // 表示关联的文件描述符，用于读写文件
91     char szFileName[256]; // 表示请求的文件名和路径，最多存储256个字符
92     char postfix[10]; // 存储文件后缀，最多存储10个字符
93     char StatusCodeReason[100]; // 存储状态码和造成的原因

```

```

94     bool permitted;           // 表示请求是否被允许
95     char *authority;         // 存储用户提供的认证信息
96     char key[1024];          // 存储密钥信息
97     SSL_CTX *ssl_ctx;        // 存储SSL上下文信息，处理加密通信
98     void *pHttpRequest;      // 指向HTTP协议的指针
99 } REQUEST, *PREQUEST;
100
101 typedef struct HTTPSTATS
102 {
103     DWORD dwRecv; //接收和发送的字节数
104     DWORD dwSend;
105 } HTTPSTATS, *PHTTPSTATS;
106
107 #endif
108

```

HTTPS 相关定义

```

1  #define HTTPSPORT 8000 // 定义了使用的端口号
2  #define METHOD_GET 0 // 定义了各种请求对应的类型， 后续完善功能此处需要添加相关定义
3  #define METHOD_HEAD 1
4
5  #define BUFSIZZ 1024 // 定义了一次IO接受的大小为1MB
6  #define ROOTCERTPEM "root-cert.pem" //定义了openssl要使用的各个文件
7  #define ROOTKEYPEM "root-key.pem"
8  #define SERVERKEYPEM "server-key.pem"
9  #define SERVERPEM "server.pem"
10 #define PASSWORD "OPENSSL" // 定义了密码
11
12 #define HTTP_STATUS_OK "200 OK" // 定义了HTTP返回的各种状态
13 #define HTTP_STATUS_CREATED "201 Created"
14 #define HTTP_STATUS_ACCEPTED "202 Accepted"
15 #define HTTP_STATUS_NOCONTENT "204 No Content"
16 #define HTTP_STATUS_MOVEDPERM "301 Moved Permanently"
17 #define HTTP_STATUS_MOVEDTEMP "302 Moved Temporarily"
18 #define HTTP_STATUS_NOTMODIFIED "304 Not Modified"
19 #define HTTP_STATUS_BADREQUEST "400 Bad Request"
20 #define HTTP_STATUS_UNAUTHORIZED "401 Unauthorized"
21 #define HTTP_STATUS_FORBIDDEN "403 Forbidden"
22 #define HTTP_STATUS_NOTFOUND "404 File can not found!"
23 #define HTTP_STATUS_SERVERERROR "500 Internal Server Error"
24 #define HTTP_STATUS_NOTIMPLEMENTED "501 Not Implemented"
25 #define HTTP_STATUS_BADGATEWAY "502 Bad Gateway"
26 #define HTTP_STATUS_UNAVAILABLE "503 Service Unavailable"
27

```

```

1  typedef int INT; // 起了一系列别名
2  typedef unsigned int UINT;
3  typedef unsigned int *PUINT; // 无符号int的指针
4  typedef unsigned long DWORD; // 双字 4字节
5  typedef unsigned int UINTE; // 无符号int
6  typedef UINT SOCKET; // 套接字， 无符号int

```

```

7  typedef unsigned long DWORD;    // 双字
8  typedef int BOOL;              // 用int表示bool
9  typedef unsigned char BYTE;    // char 表示一个字节
10 typedef unsigned short WORD;   // short 表示一个字
11 typedef float FLOAT;           // float
12 typedef FLOAT *PFLOAT;         // float的指针
13 typedef BOOL *PBOOL;           // bool的指针, 本质是int的指针
14 typedef BOOL *LPBOOL;          // 可能是为了和windowsApi的命名习惯相符
15 typedef BYTE *PBYTE;           // 字节指针
16 typedef BYTE *LPBYTE;
17 typedef int *PINT;             // int 指针
18 typedef int *LPINT;
19 typedef WORD *PWORD;
20 typedef WORD *LPWORD;
21 typedef long *LPLONG;
22 typedef DWORD *PDWORD;
23 typedef DWORD *LPDWORD;
24 typedef void *LPVOID;
25 typedef char *LPSTR;           // 字符串
26 typedef struct sockaddr *LPSOCKADDR; // 套接字地址信息
27 typedef void *HANDLE;          // 句柄
28 #define INVALID_HANDLE_VALUE (HANDLE) - 1 // 定义了一些不合法的状态
29 #define INVALID_FILE_SIZE (DWORD)0xFFFFFFFF
30 #define INVALID_SOCKET (SOCKET)(~0)
31 #define SOCKET_ERROR (-1)

```

```

1  typedef struct REQUEST //定义了一个请求结构体
2  {
3      HANDLE hExit;
4      SOCKET Socket;      // 存储socket连接
5      int nMethod;         // 表示请求所使用的方法, 对应get和head, 后续添
        加的post等应该还需要额外添加状态
6      DWORD dwRecv;        // 已经接收的字节数
7      DWORD dwSend;        // 已经发送的字节数
8      int hFile;           // 表示请求关联的文件描述符, 用于读写文件
9      char szFileName[256]; // 表示请求的文件名和路径, 最多存储256个字符
10     char postfix[10];     // 存储文件后缀, 最多存储10个字符
11     char StatusCodeReason[100]; // 存储状态码和造成的原因
12     bool permitted;       // 表示请求是否被允许
13     char *authority;      // 存储用户提供的认证信息
14     char key[1024];        // 存储密钥信息
15     SSL_CTX *ssl_ctx;     // 存储SSL上下文信息, 处理加密通信
16     void *pHttpRequest;   // 指向HTTP协议的指针
17 } REQUEST, *PREQUEST;
18
19 typedef struct HTTPSTATS // 这段代码无效, 在文中没有出现过
20 {
21     DWORD dwRecv; //接收和发送的字节数
22     DWORD dwSend;
23 } HTTPSTATS, *PHTTPSTATS;
24

```

HttpProtocol.h

完整代码

```
1  #include "common.h"
2
3  #include <map>
4  using namespace std;
5
6  class CHttpProtocol
7  {
8  public:
9      char *ErrorMsg;           // 保存错误信息
10     SOCKET m_listenSocket;     // 监听客户端链接的套接字 对应的文件描述符
11     map<char *, char *> m_typeMap; // 定义一个map, 建立文件后缀与MIME的映射
12     HANDLE m_hExit;           // 控制服务器退出
13
14     char *m_strRootDir; // web服务的根目录路径
15     UINT m_nPort;       // server服务器的端口号
16
17     BIO *bio_err;
18     // IO对象, 用来输出SSL中的错误信息
19     static char *pass;
20     // 保存密码
21     SSL_CTX *ctx;
22     // SSL上下文
23     char *initialize_ctx();
24     // 初始化SSL上下文函数, 返回字符串表示成功与否
25     char *load_dh_params(SSL_CTX *ctx, char *file);
26     // 该函数用于加载 Diffie-Hellman 参数到 SSL 上下文中, 并返回一个字符串表示加载结果。
27     static int password_cb(char *buf, int num, int rwflag, void *userdata);
28     // 获取自动获取用户密码
29
30 public:
31     CHttpProtocol(void); // 创建这个对象时自动回执行的构造函数, 在本web中完成了一些初始化工作
32     int TcpListen();      // 创建一个TCP监听, 返回和它绑定的文件描述符
33     void err_exit(char *str); // 输出错误字符串并退出
34
35     void StopHttpSrv(); // 停止HTTP服务
36     bool StartHttpSrv(); // 启动HTTP服务, 并返回bool判断失败与否
37
38     static void *ListenThread(LPVOID param); // 监听客户请求的线程
39     static void *ClientThread(LPVOID param); // 处理客户请求的线程
40
41     bool RecvRequest(PREQUEST pReq, LPBYTE pBuf, DWORD dwBufSize); // 没有使用过, 被SSLRecvRequest替代
42     int Analyze(PREQUEST pReq, LPBYTE pBuf); // 分析
43     // HTTP请求, 获取请求类型与请求的文件资源, 返回值判断成功与否
44     void Disconnect(PREQUEST pReq); // 断开与
45     // 某个请求的连接
46     void CreateTypeMap(); // 创建后
47     // 缀名与MIME类型映射
48     void SendHeader(PREQUEST pReq); // 发送
49     // HTTP头部 被替代了
```

```

40     int FileExist(PREQUEST pReq); // 检测文件是否存在
41
42     void GetCurrentTime(LPSTR lpszString); // 获得当前时间，保存在字符串中
43     bool GetLastModified(HANDLE hFile, LPSTR lpszString); // 获取上次文件的更新时间，没有使用过
44     bool GetContentType(PREQUEST pReq, LPSTR type); // 获取对应的内容，返回正确与否
45     void SendFile(PREQUEST pReq); // 发送文件
46     bool SendBuffer(PREQUEST pReq, LPBYTE pBuf, DWORD dwBufSize); // 发送数据缓冲区
47 public:
48     bool SSLRecvRequest(SSL *ssl, BIO *io, LPBYTE pBuf, DWORD dwBufSize); // SSL接受请求
49     bool SSLSendHeader(PREQUEST pReq, BIO *io); // SSL发送头部
50     bool SSLSendFile(PREQUEST pReq, BIO *io); // SSL发送文件
51     bool SSLSendBuffer(PREQUEST pReq, LPBYTE pBuf, DWORD dwBufSize); // SSL发送缓冲区
52
53 public:
54     ~CHttpProtocol(void); // 析构函数，删除对象时调用，清理SSL上下文
55     void Test(PREQUEST pReq); // 测试函数
56 };
57

```

HttpProtocol.cpp

实现了HttpProtocol.h中定义的函数

```

1  #include "common.h"
2  #include <sys/stat.h>
3  #include "HttpProtocol.h"
4
5  char *CHttpProtocol::pass = PASSWORD; // 设置密码
6  CHttpProtocol::CHttpProtocol(void)
7  {
8      bio_err = 0;
9      m_strRootDir = "/home/webServer"; // 设置web服务的目录
10     ErrorMsg = "";
11
12     ErrorMsg = initialize_ctx(); // 初始化SSL上下文，设置回调函数，设置服务器秘钥，证书，根证书
13     if (ErrorMsg == "")
14     {
15         ErrorMsg = load_dh_params(ctx, ROOTKEYPEM); // 从文件中加载dh参数
16     }
17     else
18         printf("%s \n", ErrorMsg);
19 }
20
21 CHttpProtocol::~CHttpProtocol(void)
22 {

```



```

23 // 对象被销毁时自动调用， 用于释放分配的SSL上下文
24 SSL_CTX_free(ctx);
25 }
26
27 // 初始化SSL,加载各种证书文件，设置回调方法
28 char *CHttpProtocol::initialize_ctx()
29 {
30     const SSL_METHOD *meth;
31
32     if (!bio_err) // 一开始没有设置 是0
33     {
34
35         SSL_library_init(); // 初始化OpenSSL库
36
37         SSL_load_error_strings(); // 加载SSL错误信息
38
39         bio_err = BIO_new_fp(stderr, BIO_NOCLOSE); // 创建一个输出错误信息的
bio_err
40     }
41     else
42     { // 如果在这之前已经初始化过了，再初始化就会报错
43         return "initialize_ctx() error!";
44     }
45
46     meth = SSLv23_method(); // 选择一个SSL通信协议版本
47     ctx = SSL_CTX_new(meth); // 并用这个协议创建SSL上下文， ctx包含了SSL的各种信息
48
49     // 设置SSL的自签名证书，加载失败则返回错误信息
50     if (!(SSL_CTX_use_certificate_chain_file(ctx, SERVERPEM)))
51     {
52         char *Str = "SSL_CTX_use_certificate_chain_file error!";
53         return Str;
54     }
55
56     // 设置SSL上下文密码回调函数， 用于自动获取密码
57     SSL_CTX_set_default_passwd_cb(ctx, password_cb);
58
59     // 加载服务器的私钥文件，如果错误返回信息
60     if (!(SSL_CTX_use_PrivateKey_file(ctx, SERVERKEYPEM, SSL_FILETYPE_PEM)))
61     {
62         char *Str = "SSL_CTX_use_PrivateKey_file error!";
63         return Str;
64     }
65
66     // 验证对端证书的根证书文件
67     if (!(SSL_CTX_load_verify_locations(ctx, ROOTCERTPEM, 0)))
68     {
69         char *Str = "SSL_CTX_load_verify_locations error!";
70         return Str;
71     }
72     return "";
73 }
74
75 char *CHttpProtocol::load_dh_params(SSL_CTX *ctx, char *file)
76 {
77     DH *ret = 0;

```

```

78     BIO *bio;
79
80     // 打开ROOTKEY文件失败
81     if ((bio = BIO_new_file(file, "r")) == NULL)
82     {
83         char *Str = "BIO_new_file error!";
84         return Str;
85     }
86
87     ret = PEM_read_bio_DHparams(bio, NULL, NULL, NULL); // 返回DH参数
88     BIO_free(bio); // 释放IO
89     if (SSL_CTX_set_tmp_dh(ctx, ret) < 0) // 在SSL中设置DH参数
90     {
91         char *Str = "SSL_CTX_set_tmp_dh error!";
92         return Str;
93     }
94     return ""; // 设置成功
95 }
96
97 // 把密码复制到buf中, 并返回密码长度
98 int CHttpRequest::password_cb(char *buf, int num, int rwflag, void
*userdata)
99 {
100     if ((unsigned int)num < strlen(pass) + 1)
101     {
102         return (0);
103     }
104
105     strcpy(buf, pass);
106     return (strlen(pass));
107 }
108
109 void CHttpRequest::err_exit(char *str) // 输出相关信息退出
110 {
111     printf("%s \n", str);
112     exit(1);
113 }
114
115
116 void CHttpRequest::Disconnect(PREQUEST pReq) //断开与这个请求的连接
117 {
118
119     int nRet;
120     printf("Closing socket! \r\n");
121
122     nRet = close(pReq->Socket);
123     if (nRet == SOCKET_ERROR)
124     {
125         // 关闭失败
126         printf("Closing socket error! \r\n");
127     }
128
129     // HTTPSTATS stats;
130     // stats.dwRecv = pReq->dwRecv;
131     // stats.dwSend = pReq->dwSend;
132     // SendMessage(m_hwndDlg, DATA_MSG, (UINT)&stats, NULL);

```

```

133 }
134
135 void CHttpProtocol::CreateTypeMap() // 建立后缀名与mime类型的映射，用于HTTP头部的
Content-Type字段
136 {
137
138     m_typeMap[".doc"] = "application/msword";
139     m_typeMap[".bin"] = "application/octet-stream";
140     m_typeMap[".dll"] = "application/octet-stream";
141     m_typeMap[".exe"] = "application/octet-stream";
142     m_typeMap[".pdf"] = "application/pdf";
143     m_typeMap[".ai"] = "application/postscript";
144     m_typeMap[".eps"] = "application/postscript";
145     m_typeMap[".ps"] = "application/postscript";
146     m_typeMap[".rtf"] = "application/rtf";
147     m_typeMap[".fdf"] = "application/vnd.fdf";
148     m_typeMap[".arj"] = "application/x-arj";
149     m_typeMap[".gz"] = "application/x-gzip";
150     m_typeMap[".class"] = "application/x-java-class";
151     m_typeMap[".js"] = "application/x-javascript";
152     m_typeMap[".lzh"] = "application/x-lzh";
153     m_typeMap[".lnk"] = "application/x-ms-shortcut";
154     m_typeMap[".tar"] = "application/x-tar";
155     m_typeMap[".hlp"] = "application/x-winhelp";
156     m_typeMap[".cert"] = "application/x-x509-ca-cert";
157     m_typeMap[".zip"] = "application/zip";
158     m_typeMap[".cab"] = "application/x-compressed";
159     m_typeMap[".arj"] = "application/x-compressed";
160     m_typeMap[".aif"] = "audio/aiff";
161     m_typeMap[".aifc"] = "audio/aiff";
162     m_typeMap[".aiff"] = "audio/aiff";
163     m_typeMap[".au"] = "audio/basic";
164     m_typeMap[".snd"] = "audio/basic";
165     m_typeMap[".mid"] = "audio/midi";
166     m_typeMap[".rm"] = "audio/midi";
167     m_typeMap[".mp3"] = "audio/mpeg";
168     m_typeMap[".vox"] = "audio/voxware";
169     m_typeMap[".wav"] = "audio/wav";
170     m_typeMap[".ra"] = "audio/x-pn-realaudio";
171     m_typeMap[".ram"] = "audio/x-pn-realaudio";
172     m_typeMap[".bmp"] = "image/bmp";
173     m_typeMap[".gif"] = "image/gif";
174     m_typeMap[".jpeg"] = "image/jpeg";
175     m_typeMap[".jpg"] = "image/jpeg";
176     m_typeMap[".tif"] = "image/tiff";
177     m_typeMap[".tiff"] = "image/tiff";
178     m_typeMap[".xbm"] = "image/xbm";
179     m_typeMap[".vrl"] = "model/vrml";
180     m_typeMap[".htm"] = "text/html";
181     m_typeMap[".html"] = "text/html";
182     m_typeMap[".c"] = "text/plain";
183     m_typeMap[".cpp"] = "text/plain";
184     m_typeMap[".def"] = "text/plain";
185     m_typeMap[".h"] = "text/plain";
186     m_typeMap[".txt"] = "text/plain";
187     m_typeMap[".rtx"] = "text/richtext";

```

```

188     m_typeMap[".rtf"] = "text/richtext";
189     m_typeMap[".java"] = "text/x-java-source";
190     m_typeMap[".css"] = "text/css";
191     m_typeMap[".mpeg"] = "video/mpeg";
192     m_typeMap[".mpg"] = "video/mpeg";
193     m_typeMap[".mpe"] = "video/mpeg";
194     m_typeMap[".avi"] = "video/msvideo";
195     m_typeMap[".mov"] = "video/quicktime";
196     m_typeMap[".qt"] = "video/quicktime";
197     m_typeMap[".shtml"] = "wwwserver/html-ssi";
198     m_typeMap[".asa"] = "wwwserver/isapi";
199     m_typeMap[".asp"] = "wwwserver/isapi";
200     m_typeMap[".cfm"] = "wwwserver/isapi";
201     m_typeMap[".dbm"] = "wwwserver/isapi";
202     m_typeMap[".isa"] = "wwwserver/isapi";
203     m_typeMap[".plx"] = "wwwserver/isapi";
204     m_typeMap[".url"] = "wwwserver/isapi";
205     m_typeMap[".cgi"] = "wwwserver/isapi";
206     m_typeMap[".php"] = "wwwserver/isapi";
207     m_typeMap[".wsgi"] = "wwwserver/isapi";
208 }
209
210 int CHttpProtocol::TcpListen() // 创建套接字，并与网络地址绑定，返回对应的文件描述符
sock
211 {
212     int sock;
213     struct sockaddr_in sin; //表示Ipv4网络地址的结构体
214
215     if ((sock = socket(PF_INET, SOCK_STREAM, 0)) < 0) // 创建新的套接字，返回对
应的文件描述符
216         err_exit("Couldn't make socket");
217
218     memset(&sin, 0, sizeof(sin));
219     sin.sin_addr.s_addr = INADDR_ANY; // 绑定到所有网络接口
220     sin.sin_family = PF_INET; // 使用ipv4地址簇
221     sin.sin_port = htons(8000); // 监听端口号为8000
222
223     if (bind(sock, (struct sockaddr *)&sin, sizeof(struct sockaddr)) < 0) //
sock对应的文件描述符与sockaddr绑定在一起
224         err_exit("Couldn't bind");
225     listen(sock, 5); // 设置请求队列的长度
226     // printf("TcpListen Ok\n");
227
228     return sock;
229 }
230
231 bool CHttpProtocol::SSLRecvRequest(SSL *ssl, BIO *io, LPBYTE pBuf, DWORD
dwBufSize)
232 {
233     // printf("SSLRecvRequest \n");
234     char buf[BUFSIZZ]; // 分多次接收，一次最多接收1MB，pBUF只有4MB
235     int r, length = 0;
236
237     memset(buf, 0, BUFSIZZ); // 清空buf
238     while (1)
239     {

```

```

240     r = BIO_gets(io, buf, BUFSIZZ - 1); // r为读到的字符数
241     // printf("r = %d\r\n",r);
242     switch (SSL_get_error(ssl, r)) // 获取错误码
243     {
244     case SSL_ERROR_NONE: // 没有错误, 将获取的字符串拼接到pubf上, 同时len+=r,
更新已经读取到的字符串
245         memcpy(&pBuf[length], buf, r);
246         length += r;
247         // printf("Case 1... \r\n");
248         break;
249     default:
250         // printf("Case 2... \r\n");
251         break;
252     }
253     // 判断是否读取到了http的头部结束标志
254     if (!strcmp(buf, "\r\n") || !strcmp(buf, "\n"))
255     {
256         printf("IF...\r\n");
257         break;
258     }
259 }
260 // 添加字符表示字符串结束
261 pBuf[length] = '\0';
262 return true;
263 }
264 bool CHttpProtocol::StartHttpSrv() // 启动http服务
265 {
266     CreateTypeMap(); // 创建映射
267
268     printf("*****Server starts***** \n");
269
270     pid_t pid;
271     m_listenSocket = TcpListen(); // 开启tcp监听
272
273     pthread_t listen_tid;
274     pthread_create(&listen_tid, NULL, &ListenThread, this); // 创建监听线程
275 }
276
277 void *CHttpProtocol::ListenThread(LPVOID param)
278 {
279     printf("Starting ListenThread... \n");
280
281     CHttpProtocol *pHttpProtocol = (CHttpProtocol *)param;
282
283     SOCKET socketClient; //客户端的套接字文件描述符
284     pthread_t client_tid;
285     struct sockaddr_in SockAddr; // 客户端地址结构体
286     PREQUEST pReq;
287     socklen_t nLen; // 客户端地址结构体大小
288     DWORD dwRet;
289
290     while (1)
291     {
292         // printf("while!\n");
293         nLen = sizeof(SockAddr);
294

```

```

295     socketClient = accept(pHttpProtocol->m_listenSocket,
(LPSOCKADDR)&SockAddr, &nLen); // 接收客户端的连接请求, 返回客户端的socket
296     // printf("%s ",inet_ntoa(SockAddr.sin_addr));
297     if (socketClient == INVALID_SOCKET) // 如果返回-1, 表示操作失败, 退出
298     {
299         printf("INVALID_SOCKET !\n");
300         break;
301     }
302     pReq = new REQUEST; // 构造客户端的请求
303     // pReq->hExit = pHttpProtocol->m_hExit;
304     pReq->Socket = socketClient; //客户端套接字
305     pReq->hFile = -1;
306     pReq->dwRecv = 0;
307     pReq->dwSend = 0;
308     pReq->pHttpProtocol = pHttpProtocol;
309     pReq->ssl_ctx = pHttpProtocol->ctx; // 所有连接使用的都是同一个ctx? ?
310
311     // printf("New request");
312     pthread_create(&client_tid, NULL, &ClientThread, pReq); // 一旦有新的
客户端请求, 就建立一个线程处理
313 } // while
314
315     return NULL;
316 }
317
318 void *CHttpProtocol::ClientThread(LPVOID param) // 处理客户端请求的线程
319 {
320     printf("Starting ClientThread... \n");
321     int nRet;
322     SSL *ssl;
323     BYTE buf[4096];
324     BIO *sbio, *io, *ssl_bio;
325     PREQUEST pReq = (PREQUEST)param;
326     CHttpProtocol *pHttpProtocol = (CHttpProtocol *)pReq->pHttpProtocol;
327     // pHttpProtocol->CountUp(); // ????
328     SOCKET s = pReq->Socket;
329
330     sbio = BIO_new_socket(s, BIO_NOCLOSE); // 为对应的socket创建io对象, 处理输入
输出
331     ssl = SSL_new(pReq->ssl_ctx); // 用上下文ctx创建一个新的ssl连接
332     SSL_set_bio(ssl, sbio, sbio); // 将ssl与bio关联起来
333
334     nRet = SSL_accept(ssl); // 完成ssl的握手过程
335
336     if (nRet <= 0) // 握手失败
337     {
338         pHttpProtocol->err_exit("SSL_accept()error! \r\n");
339         // return 0;
340     }
341
342     io = BIO_new(BIO_f_buffer()); // 创建缓冲区 io
343     //
344     ssl_bio = BIO_new(BIO_f_ssl()); // 创建ssl io
345     //
346     BIO_set_ssl(ssl_bio, ssl, BIO_CLOSE); // 绑定ssl和ssl_bio, ssl_bio的读写
都是用这个ssl进行加密

```

```

347     BIO_push(io, ssl_bio); // 形成数据传输链，对io进行操作时候，都
    会经过ssl_bio进行加密
348
349
350     printf("*****\r\n");
351     if (!pHttpProtocol->SSLRecvRequest(ssl, io, buf, sizeof(buf))) // 如果没
    有读取到请求头部相关信息
352     {
353         // 接收错误
354         pHttpProtocol->err_exit("Receiving SSLRequest error!! \r\n");
355     }
356     else
357     {
358         printf("Request received!! \n"); // 收到了，并且输出相关信息
359         printf("%s \n", buf);
360         // return 0;
361     }
362     nRet = pHttpProtocol->Analyze(pReq, buf); // 分析请求的信息，将请求路径保存到
    对应的pReq-filename中
363     if (nRet) // 为真则失败
364     {
365         // 断开对应的连接?
366         pHttpProtocol->Disconnect(pReq);
367         delete pReq;
368         pHttpProtocol->err_exit("Analyzing request from client
    error!!\r\n");
369     }
370
371     if (!pHttpProtocol->SSLSendHeader(pReq, io)) // 发送响应头
372     {
373         pHttpProtocol->err_exit("Sending fileheader error!\r\n");
374     }
375     BIO_flush(io);
376
377     // 如果是get方法
378     if (pReq->nMethod == METHOD_GET)
379     {
380         printf("Sending.....\n");
381         if (!pHttpProtocol->SSLSendFile(pReq, io)) // 发送对应文件， 失败就返回
382         {
383             return 0;
384         }
385     }
386     printf("File sent!!");
387     // pHttpProtocol->Test(pReq);
388     pHttpProtocol->Disconnect(pReq); // 文件成功发送，断开连接
389     delete pReq; // 释放对应的请求
390     SSL_free(ssl); // 释放ssl
391     return NULL;
392 }
393
394 int CHttpProtocol::Analyze(PREQUEST pReq, LPBYTE pBuf) // 解析客户端发送的请求函
    数 返回0成功 返回1失败
395 {
396
397     char szSeps[] = " \n";

```

```

398     char *cpToken;
399
400     if (strstr((const char *)pBuf, "..") != NULL) // 如果请求路径中可能存在路径遍
历史攻击，直接干掉
401     {
402         strcpy(pReq->StatuCodeReason, HTTP_STATUS_BADREQUEST);
403         return 1;
404     }
405
406     // 提取下一个片段，请求类型
407     cpToken = strtok((char *)pBuf, szSeps);
408     if (!strcmp(cpToken, "GET")) // 判断是否是get请求
409     {
410         pReq->nMethod = METHOD_GET;
411     }
412     else if (!strcmp(cpToken, "HEAD")) // 判断是否是HEAD请求
413     {
414         pReq->nMethod = METHOD_HEAD;
415     }
416     else
417     {
418         strcpy(pReq->StatuCodeReason, HTTP_STATUS_NOTIMPLEMENTED); // 都不
是，则不支持
419         return 1;
420     }
421
422     // 请求的URI
423     cpToken = strtok(NULL, szSeps);
424     if (cpToken == NULL) // 获取失败，返回错误
425     {
426         strcpy(pReq->StatuCodeReason, HTTP_STATUS_BADREQUEST);
427         return 1;
428     }
429
430     strcpy(pReq->szFileName, m_strRootDir); // 在 HTTP 服务器中，客户端请求的
URI 通常表示客户端请求的资源路径，但是这个路径是相对于服务器根目录的。因此，在处理请求时，
需要将客户端请求的 URI 转换为服务器上的绝对路径，以便服务器能够定位到正确的资源文件。
431     if (strlen(cpToken) > 1)
432     {
433         strcat(pReq->szFileName, cpToken); // 继续拼接url
434     }
435     else
436     {
437         strcat(pReq->szFileName, "/index.html"); // 否则默认将这个添加到末尾
438     }
439     printf("%s\r\n", pReq->szFileName); // 输出对应的请求文件路径
440
441     return 0;
442 }
443
444 int CHttpProtocol::FileExist(PREQUEST pReq) // 查看请求的文件是否存在 0没找到， 1
找到了 并设置对应的文件描述符
445 {
446     pReq->hFile = open(pReq->szFileName, O_RDONLY); // 只读形式打开对应文件
447
448     if (pReq->hFile == -1) // 如果不存在

```



```

449     {
450         strcpy(pReq->StatuCodeReason, HTTP_STATUS_NOTFOUND); // 文件不存在错误
451         printf("open %s error\n", pReq->szFileName);
452         return 0;
453     }
454     else
455     {
456         // printf("hFile\n"); // 找到了
457         return 1;
458     }
459 }
460 void CHttpProtocol::Test(PREQUEST pReq)
461 {
462     struct stat buf;
463     long fl;
464     if (stat(pReq->szFileName, &buf) < 0)
465     {
466         err_exit("Getting filesize error!!\r\n");
467     }
468     fl = buf.st_size;
469     printf("Filesize = %d \r\n", fl);
470 }
471
472 void CHttpProtocol::GetCurrentTime(LPSTR lpszString) // 获取当前时间到 传入的字符串中
473 {
474
475     char *week[] = {
476         "Sun,",
477         "Mon,",
478         "Tue,",
479         "Wed,",
480         "Thu,",
481         "Fri,",
482         "Sat,",
483     };
484
485     char *month[] = {
486         "Jan",
487         "Feb",
488         "Mar",
489         "Apr",
490         "May",
491         "Jun",
492         "Jul",
493         "Aug",
494         "Sep",
495         "Oct",
496         "Nov",
497         "Dec",
498     };
499
500     struct tm *st;
501     long ct;
502     ct = time(&ct);
503     st = (struct tm *)localtime(&ct);

```

```

504
505     sprintf(lpszString, "%s %02d %s %d %02d:%02d:%02d GMT", week[st-
>tm_wday], st->tm_mday, month[st->tm_mon],
506         1900 + st->tm_year, st->tm_hour, st->tm_min, st->tm_sec);
507 }
508
509 bool CHttpProtocol::GetContentType(PREQUEST pReq, LPSTR type)
510 {
511
512     char *cpToken;
513     cpToken = strstr(pReq->szFileName, "."); // 查找第一个.后面的字符串, 文件后缀
名
514     strcpy(pReq->postfix, cpToken); // 把文件后缀名保存到postfix中
515
516     map<char *, char *>::iterator it = m_typeMap.find(pReq->postfix); // 通过
后缀名找到对应的MIME类型
517     if (it != m_typeMap.end())
518     {
519         sprintf(type, "%s", (*it).second); // 格式化保存到type中
520     }
521     return 1;
522 }
523
524 bool CHttpProtocol::SSLSendHeader(PREQUEST pReq, BIO *io)
525 {
526     char Header[2048] = " ";
527     int n = FileExist(pReq); // 判断是否存在对应请求的文件
528     if (!n) // 如果不存在
529     {
530         err_exit("The file requested doesn't exist!");
531     }
532
533     char curTime[50];
534     CHttpProtocol::GetCurrentTime(curTime); // 获取当前时间
535
536     struct stat buf;
537     long length;
538     if (stat(pReq->szFileName, &buf) < 0) // 获取文件的状态信息
539     {
540         err_exit("Getting filesize error!!\r\n");
541     }
542     length = buf.st_size; // 获取文件字节数
543
544     char ContentType[50] = " "; // 存类型
545     GetContentType(pReq, (char *)ContentType); // 获取对应的MIME类型保存到
ContentType
546     // 将响应头信息写到Header中
547     sprintf((char *)Header, "HTTP/1.1 %s\r\nDate: %s\r\nServer:
%s\r\nContent-Type: %s\r\nContent-Length: %d\r\n\r\n",
548         HTTP_STATUS_OK,
549         curTime, // Date
550         "Villa Server 192.168.176.139", // Server"My Https Server"
551         ContentType, // Content-Type
552         length); // Content-length
553

```

```

554     if (BIO_write(io, Header, strlen(Header)) <= 0) // 将这个信息写入SSL连接的
BIO中, 失败返回false
555     {
556         return false;
557     }
558     BIO_flush(io); // 刷新BIO, 确保信息已经发送到对方
559     printf("SSLSendHeader successfully!\n");
560     return true;
561 }
562
563 bool CHttpRequest::SSLSendFile(PREQUEST pReq, BIO *io)
564 {
565     // printf("%s\n", pReq->szFileName);
566     int n = FileExist(pReq);
567
568     if (!n)
569     {
570         err_exit("The file requested doesn't exist!");
571     }
572
573     static char buf[2048];
574     DWORD dwRead;
575     BOOL fRet;
576     int flag = 1, nReq;
577
578     while (1) // 开始发送
579     {
580
581         fRet = read(pReq->hFile, buf, sizeof(buf)); // 每次读取2MB
582         // printf("%d,%d\n", fRet, pReq->hFile);
583         if (fRet < 0) // 如果读取失败
584         {
585             // printf("!fRet\n");
586             static char szMsg[512];
587             sprintf(szMsg, "%s", HTTP_STATUS_SERVERERROR); // 返回错误信息
588
589             // if((nReq = BIO_puts(io, szMsg)) <= 0)
590             if ((nReq = BIO_write(io, szMsg, strlen(szMsg))) <= 0) // 输出错
误信息 终止循环
591             {
592                 err_exit("BIO_write() error!\n");
593             }
594             BIO_flush(io);
595             break;
596         }
597
598         // 如果读取到文件末尾 则完成
599         if (fRet == 0)
600         {
601             printf("complete \n");
602             break;
603         }
604
605         // if(BIO_puts(io, buf) <= 0)
606         if (BIO_write(io, buf, fRet) <= 0) // 读到的数据写过去, 并判断是否成功
607         {

```

```

608         if (!BIO_should_retry(io)) // 判断是否可以重试
609         {
610             printf("BIO_write() error!\r\n");
611             break;
612         }
613     }
614     BIO_flush(io);
615
616     pReq->dwSend += fRet; // 设置本次读取的字节数
617 }
618
619 if (close(pReq->hFile) == 0) // 关闭对应文件
620 {
621     pReq->hFile = -1;
622     return true;
623 }
624 else // ❖❖❖❖
625 {
626     err_exit("Closing file error!");
627 }
628 }
629

```

二、可调整的地方

重复定义

代码中存在重复定义的内容。

```

typedef int INT;
typedef unsigned int UINT;
typedef unsigned int *PUINT; //
typedef unsigned long DWORD; //
typedef unsigned int UINT; //
typedef UINT SOCKET; //

```

无用内容

代码中存在一些有没有使用过的内容，以及一些被SSL替代掉的函数。

```

typedef struct HTTPSTATS
{
    DWORD dwRecv; //接收和发送的字节数
    DWORD dwSend;
} HTTPSTATS, *PHTTPSTATS;

```

```

bool RecvRequest(PREQUEST pReq, LPBYTE pBuf, DWORD dwBufSize); // 没有使用过，被SSLRecvRequest
int Analyze(PREQUEST pReq, LPBYTE pBuf); // 分析HTTP请求，返回一个整数
void Disconnect(PREQUEST pReq); // 没有连接？断开连接？
void CreateTypeMap(); // 创建类型映射
void SendHeader(PREQUEST pReq); // 发送HTTP头部
int FileExist(PREQUEST pReq); // 检测文件是否存在

void GetCurrentTime(LPSTR lpszString); // 获得当前时间，当前字符串的对应
bool GetLastModified(HANDLE hFile, LPSTR lpszString); // 获取上次文件的更新时间
bool GetContentType(PREQUEST pReq, LPSTR type); // 获取对应的内容，返回正确与否
void SendFile(PREQUEST pReq); // 发送文件
bool SendBuffer(PREQUEST pReq, LPBYTE pBuf, DWORD dwBufSize); // 发送数据缓冲区
public:
bool SSLRecvRequest(SSL *ssl, BIO *io, LPBYTE pBuf, DWORD dwBufSize); // SSL接受请求
bool SSLSendHeader(PREQUEST pReq, BIO *io); // SSL发送头部
bool SSLSendFile(PREQUEST pReq, BIO *io); // SSL发送文件
bool SSLSendBuffer(PREQUEST pReq, LPBYTE pBuf, DWORD dwBufSize); // SSL发送缓冲区

```

替代掉了，没有用过

文件不存在直接结束服务器

从下面代码中可知，当客户端请求的文件不存在时，会执行exit(1)直接退出进程，这显然是非常不合理的，合理的做法应当是返回错误信息，而不是直接干掉服务器。

不仅仅是这个，该Web服务器在许多情况下都选择直接干掉服务器，而不是进行错误处理。当然，其他的错误大都是服务器问题，干掉服务器情有可原。**请求文件不存在这是客户端可以控制的，利用这个攻击者可以轻易干掉服务器。**

```

1      int n = FileExist(pReq); // 判断是否存在对应请求的文件
2      if (!n) // 如果不存在
3      {
4          err_exit("The file requested doesn't exist!");
5      }
6
7
8      void CHttpProtocol::err_exit(char *str) // 输出相关信息退出
9      {
10         printf("%s \n", str);
11         exit(1);
12     }

```

请求路径中存在.. 直接结束服务器

如果检测到..，则存在路径遍历攻击，会结束服务器，这也合理，返回错误信息即可。

```

1
2      if (strstr((const char *)pBuf, "..") != NULL) // 如果请求路径中可能存在路径遍
        历攻击，直接干掉
3      {
4          strcpy(pReq->StatusCodeReason, HTTP_STATUS_BADREQUEST);
5          return 1;
6      }

```

Content-Type类型无法显示bug

经过测试我发现，不论存在映射的文件类型，还是不存在映射的文件类型，Content-type都无法正常显示。

```
/home/WebServer/index.html  
postfix==.html  
MIME==(null)
```

通过调试我发现: 对于存在类型，使用Map查找也只能得到null，但是我通过打印能得到对应的值：

```
key:.h,value:text/plain  
key:.txt,value:text/plain  
key:.rtx,value:text/richtext  
key:.java,value:text/x-java-source  
key:.css,value:text/css  
key:.mpeg,value:video/mpeg  
key:.mpg,value:video/mpeg  
key:.mpe,value:video/mpeg  
key:.avi,value:video/msvideo  
key:.mov,value:video/quicktime  
key:.qt,value:video/quicktime  
key:.shtml,value:wwwserver/html-ssi  
key:.asa,value:wwwserver/isapi  
key:.asp,value:wwwserver/isapi  
key:.cfm,value:wwwserver/isapi  
key:.dbm,value:wwwserver/isapi  
key:.isa,value:wwwserver/isapi  
key:.plx,value:wwwserver/isapi  
key:.url,value:wwwserver/isapi  
key:.cgi,value:wwwserver/isapi  
key:.php,value:wwwserver/isapi  
key:.wsgi,value:wwwserver/isapi
```

三、WebServer的测试

正常GET请求——默认路径 /WebServer

Web捕获到的请求信息

```
Request received!!  
GET / HTTP/1.1  
User-Agent: Apifox/1.0.0 (https://apifox.com)  
Accept: */*  
Host: 192.168.0.100:8000  
Accept-Encoding: gzip, deflate, br  
Connection: keep-alive
```

可以看到请求类型为：GET

请求路径为：/，默认为 /index.html,并且相对路径会在服务端被拼接成绝对路径

```

1      strcpy(pReq->szFileName, m_strRootDir); // 在 HTTP 服务器中，客户端请求的
      URI 通常表示客户端请求的资源路径，但是这个路径是相对于服务器根目录的。因此，在处理请求时，
      需要将客户端请求的 URI 转换为服务器上的绝对路径，以便服务器能够定位到正确的资源文件。
2      if (strlen(cpToken) > 1)
3      {
4          strcat(pReq->szFileName, cpToken); // 继续拼接url
5      }
6      else
7      {
8          strcat(pReq->szFileName, "/index.html"); // 否则默认将这个添加到末尾
9      }

```

返回的信息

GET
https://192.168.0.100:8000/

Params
Body
Headers 7
Cookies
Auth
前置操作
后置操作
设置

Query 参数

参数名	参数值
添加参数	

Body
Cookie
Header 4
控制台
实际请求
分享

Pretty
Raw
Preview
Visualize
Text
utf8

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="utf-8">
5  <title>文档标题</title>
6  <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.1/jquery.min.js"></script>
7  </head>
8  <link rel="icon" type="image/x-icon" href="favicon.ico" />
9  <body>
10     <h1>我的第一个HTML页面</h1>
11     <p>我的第一个段落。</p>
12 </body>
13 </html>
14

```

请求其他文件——favicon.ico

Web捕获到的请求信息

请求文件的地址变成了 /favicon.ico, 服务器拼接了完整的地址 /home/webServer/favicon.ico


```
Request received!!  
GET /favicon.ico HTTP/1.1  
User-Agent: Apifox/1.0.0 (https://apifox.com)  
Accept: */*  
Host: 192.168.0.100:8000  
Accept-Encoding: gzip, deflate, br  
Connection: keep-alive
```

```
/home/WebServer/favicon.ico  
SSLSendHeader successfully!  
Sending.....  
complete  
File sent!!Closing socket!
```

返回额信息

[Body](#) [Cookie](#) [Header 4](#) [控制台](#) [实际请求](#) [分享](#)

图片: image/x-icon [下载](#) [仍以文本查看](#)



类型缺失bug

可以看到，因为请求的类型在Map中没有建立映射，所以 Content-Type 这一栏为空

对于没有建立相关映射的文件，是否应该定义为非法文件，不允许请求呢？

[Body](#) [Cookie](#) [Header 4](#) [控制台](#) [实际请求](#) [分享](#)

名称	值
Date	Fri, 10 May 2024 23:43:21 GMT
Server	Villa Server 192.168.176.139
Content-Type	
Content-Length	4286

四、报文分析

一次完整请求的报文交互过程

TCP三次握手，四次挥手

可以看到截图中前三个报文就是TCP三次握手。

后续发送完文件**只有客户端对服务端进行断开连接，没有服务端断开连接的报文**，可能得原因是：服务端发送完信息后，发送了断开连接的请求报文，在收到确认后，关闭了SSL，所以后续可以看到客户端的断开请求报文。

49	9.234851	192.168.0.103	192.168.0.100	TCP	74	48934 → 8000 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=706118551 TSecr=0 WS=128
50	9.234971	192.168.0.100	192.168.0.103	TCP	74	8000 → 48934 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM TSval=1260198219 TSecr=706118551 WS=128
51	9.235000	192.168.0.103	192.168.0.100	TCP	66	48934 → 8000 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=706118551 TSecr=1260198219
52	9.237373	192.168.0.103	192.168.0.100	TLSv1.3	639	Client Hello
53	9.237459	192.168.0.100	192.168.0.103	TCP	66	8000 → 48934 [ACK] Seq=1 Ack=574 Win=64640 Len=0 TSval=1260198221 TSecr=706118554
54	9.243596	192.168.0.100	192.168.0.103	TLSv1.3	291	Server Hello, Change Cipher Spec, Application Data, Application Data
55	9.243785	192.168.0.103	192.168.0.100	TCP	66	48934 → 8000 [ACK] Seq=574 Ack=226 Win=64128 Len=0 TSval=706118560 TSecr=1260198227
56	9.244715	192.168.0.103	192.168.0.100	TLSv1.3	130	Change Cipher Spec, Application Data
57	9.244829	192.168.0.103	192.168.0.100	TLSv1.3	289	Application Data
58	9.245140	192.168.0.103	192.168.0.100	TLSv1.3	536	Application Data
59	9.245273	192.168.0.100	192.168.0.103	TLSv1.3	220	Application Data
60	9.245347	192.168.0.100	192.168.0.103	TLSv1.3	418	Application Data
61	9.246256	192.168.0.103	192.168.0.100	TCP	66	48934 → 8000 [ACK] Seq=1108 Ack=956 Win=64128 Len=0 TSval=706118563 TSecr=1260198229
62	9.246643	192.168.0.103	192.168.0.100	TLSv1.3	90	Application Data
63	9.246726	192.168.0.100	192.168.0.103	TCP	60	8000 → 48934 [RST] Seq=956 Win=0 Len=0
64	9.246732	192.168.0.103	192.168.0.100	TCP	66	48934 → 8000 [FIN, ACK] Seq=1132 Ack=956 Win=64128 Len=0 TSval=706118563 TSecr=1260198229
65	9.246760	192.168.0.100	192.168.0.103	TCP	60	8000 → 48934 [RST] Seq=956 Win=0 Len=0

完整的SSL过传输过程

[ssl报文分析](#)，详细记录了Openssl报文各种字段的各个信息

52	9.237373	192.168.0.103	192.168.0.100	TLSv1.3	639	Client Hello
54	9.243596	192.168.0.100	192.168.0.103	TLSv1.3	291	Server Hello, Change Cipher Spec, Application Data, Application Data
56	9.244715	192.168.0.103	192.168.0.100	TLSv1.3	130	Change Cipher Spec, Application Data
57	9.244829	192.168.0.100	192.168.0.103	TLSv1.3	289	Application Data
58	9.245140	192.168.0.103	192.168.0.100	TLSv1.3	536	Application Data
59	9.245273	192.168.0.100	192.168.0.103	TLSv1.3	220	Application Data
60	9.245347	192.168.0.100	192.168.0.103	TLSv1.3	418	Application Data

五、后续工作

因为有很多组同学选择了这个选题，并且扩展实验都选择了ddos攻击，并考虑到我一个人效率不够，所以选择更改我的策略。

- 放弃ddos攻击这个实验，很多小组做这个，学习他们的思路以及方法
- 修复源码中一些不合逻辑的地方，以及一些上面提到的bug
- 并对代码进行一些详细的注释，可以将其提供给后续选课的同学
- 完善Post请求功能，客户端像服务端提供信息

POST功能完善

添加对应的定义

```
#define HTTPSPORT 8000
#define METHOD_GET 0
#define METHOD_HEAD 1
#define METHOD_POST 2
```

添加对应判断

```
// 提取下一个片段, 请求类型
cpToken = strtok((char *)pBuf, szSeps);
if (!strcmp(cpToken, "GET")) // 判断是否是get请求
{
    pReq->nMethod = METHOD_GET;
}
else if (!strcmp(cpToken, "HEAD")) // 判断是否是HEAD请求
{
    pReq->nMethod = METHOD_HEAD;
}
else if (!strcmp(cpToken, "POST"))
{
    pReq->nMethod = METHOD_POST;
}
else
{
    strcpy(pReq->StatusCodeReason, HTTP_STATUS_NOTIMPLEMENTED); // 都不是, 则不支持
    return 1;
}
```

修改bool CHttpRequest::SSLRecvRequest逻辑

我的实现逻辑是这样的, 课程组提供的代码在 读取到请求头后会直接退出, 我的修改逻辑就是, 添加一个 flag 标识, flag==true 表示读请求体, flag==false 表示读请求头

定义个 payload[] 接收请求体内容, payLength 表示请求体的长度。

```
char payload[BUFSIZZ];
int r, length=0, payLength=0;
bool flag = false;
memset(payload, 0, BUFSIZZ); //初始化缓冲区
memset(payload, 0, BUFSIZZ);
```

核心逻辑如下:

```
1 while(1)
2     {
3         r = BIO_gets(io, buf, BUFSIZZ-1);
4         switch(SSL_get_error(ssl, r))
5         {
6             case SSL_ERROR_NONE:
7                 if (!flag){ // 读取请求头
8                     memcpy(&pBuf[length], buf, r);
9                     length += r;
10                }
11                else { //读取请求体
12                    memcpy(&payload[payLength], buf, r);
13                    payLength += r;
14                }
15
16                break;
17                default:
18                    break;
19            }
20            // 直到读到代表HTTP头部结束的空行
21            if(!strcmp(buf, "\r\n") || !strcmp(buf, "\n") || r==0)
```

服务端展示效果

客户端请求

Content-Type无法显示Bug修复

Body	Cookie	Header 4	控制台	实际请求	分享
名称	值				
Date	Fri, 10 May 2024 23:43:21 GMT				
Server	Villa Server 192.168.176.139				
Content-Type					
Content-Length	4286				

问题溯源

```
map<char *, char *>::iterator it = m_typeMap.find(pReq->postfix);
printf("is equal ? ans=%d\n", strcmp(pReq->postfix, ".html"));
if(it != m_typeMap.end())
{
    sprintf(type,"%s",(*it).second);
    printf("ContentType ==%s\n",(*it).second);
}
printf("ContentType==%s\n",type);
```

在上面这份代码中，输出 type 发现是空的，也就是主要问题在于 type 字段根本没有值，但是根据代码来看 它不应该没有值，这很反常。

解决方法

在尝试了多次之后，我还是没有找到问题的根源错误，采用了治标不治本的方法。

我使用了暴力遍历的方法来处理，复杂度升高，但是达到了我想要的效果

```
for (auto it = m_typeMap.begin();it != m_typeMap.end();it++){
    if (!strcmp(pReq->postfix,(*it).first)) {
        strcpy(type,(*it).second);
        break;
    }
}
```

效果展示

Name	Value
Date	Wed, 22 May 2024 23:46:24 GMT
Server	Villa Server 192.168.176.139
Content-Type	text/html
Content-Length	330

入口函数冗余分析

```
1 // SSL *ssl;
2 // BYTE buf[4096];
3 // BIO *io;
4 // bool bRet;
5 // bRet = MyHttpObj.SSLRecvRequest(ssl,io,buf,sizeof(buf));
6 // if(!bRet)
7 // {
8 //     MyHttpObj.err_exit("Receiving request error! \n");
9 // }
10 // else
11 // {
12 //     printf("Request received!! \n");
13 //     printf("%s \n",buf);
14 // }
15
```

看完源代码，我一直不明白 这段代码的意义是什么，但是当我注释掉，程序无法正常运行。

会直接结束，如下图所示：

```
ubuntu@ubuntu:~/Desktop/2024_cyber_security/SSL实验/SSL$ ./MyWebServer
*****Server starts*****
Starting ListenThread...
while!
ubuntu@ubuntu:~/Desktop/2024_cyber_security/SSL实验/SSL$ ./MyWebServer
```

调研以及分析后，我发现，这段代码唯一作用就是保证程序不结束，因为 `SSLRecvRequest` 函数中有死循环，所以只需要改成这样即可：

```
int main()
{
    CHttpProtocol MyHttpObj;
    MyHttpObj.StartHttpSrv();
    while(1){
    }
```

原因

如果主程序结束，那么我们在主程序中使用的 `MyHttpObj` 这个协议实例就会被回收。

而其他线程都使用了这个实例，在 `该实例被回收后` 也会异常退出。所以我们需要保证程序不结束，也就是保证该实例不被回收。

从下面图片可以看出，主函数结束后，析构函数被调用，对象被回收

```
1 CHttpProtocol::~~CHttpProtocol(void)
2 {
3     // 释放SSL上下文环境
4     SSL_CTX_free(ctx);
5     printf("Object is over!\n");
6 }
```

```
ubuntu@ubuntu:~/Desktop/2024_cyber_security/SSL实验/SSL$ ./MyWebServer
*****Server starts*****
Object is over!
Starting ListenThread...
while!
INVALID_SOCKET !
```

非法请求

请求文件不存在

课程组代码中，若请求文件不存在，则直接关闭服务器，这样的做法是不太合理的，所以我对其进行修改，如果请求的文件不存在，则返回一个 `error.html`

```

int CHttpProtocol::FileExist(PREQUEST pReq)
{
    pReq->hFile = open(pReq->szFileName, O_RDONLY);
    // 如果文件不存在, 则返回出错信息
    if (pReq->hFile == -1)
    {
        strcpy(pReq->StatusCodeReason, HTTP_STATUS_NOTFOUND);
        printf("The file requested doesn't exist!\n");
        strcpy(pReq->szFileName, "/home/WebServer/error.html");
        pReq->hFile = open(pReq->szFileName, O_RDONLY);
        printf("pReq->hFile==%d\n", pReq->hFile);
    }
    return 1;
}

```



Error: File Not Found or Invalid Request Method

The requested file does not exist on the server or the request method used is invalid.

请求文件的路径中存在 ..

这也算是一个非法请求, 应该返回错误信息, 而不是干掉服务器, 所以我对其进行了处理, 检测路径中是否有 .., 如果有, 则返回 error.html

```

454     printf("%s\r\n", pReq->szFileName);
455     if (strstr(pReq->szFileName, "..") != NULL) {
456         strcpy(pReq->szFileName, "/home/WebServer/error.html");
457     }
458     return 0;
459 }

```

总结

至此, 我计划做的事情告一段落。开学时没想过这学期会这么忙, 大部分精力都花在准备实习上了, 本来想使用 hping3 工具来一个 DDOS 测试, 但是奈何一个人精力不够, 马上就要开始实习了, 再加上很多同学都选择了 DDOS 测试, 应该也不差我一个人的实现方式, 所以就到此打住。

完成的工作

- 在提交的文档中, 对源码进行了较为详细的注释
- 完成了 POST 请求功能, 在读懂源码后, 其余功能大同小异
 - 需要注意的是 POST 携带内容时, 需要多打一个换行, 这算是代码的小缺陷
- 解决了服务器返回的 Response 无法显示 Content-Type 的问题
- 解决了 请求文件不存在会直接结束服务器 问题
- 解决了 客户端请求路径有 .. 时结束服务器 问题