NetSecProj

Michael Chess [MSC2209]
Elias Cottingham [EBC2133]
Sammy Tbeile [ST2918]
Group 1

COMS 4180 Network Security
Group Project Part 1

NOTE: CHUNK should be 1440 for accurate approximation of MTU of ethernet. This is because the MTU for ethernet is 1518 less the ethernet header is 1500 less the TCP header is minimum 1440 (TCP header is variable).

Installation Guide:

```
Make is required as is a c compiler.
On fresh Ubuntu systems "sudo apt-get install build-essential" may be required.
Otherwise just 'make' should do the trick.
Additionally openssl and libopenssl are required
On fresh Ubuntu systems: "sudo apt-get install openssl libssl-dev"
```

Usage:
make will result in two executables: server_handler and client.
server_handler should be run with:
./server_handler
Example: [Fake filenames]
./server_handler 10001 ./ftp_dir test_ids_sigs.txt ids_file.txt
client should be run with:
./client
Example: [Fake filenames]
./client 10001 127.0.0.1 ./client_dir

```
Using the ftp server. Client side:
    Once the client has started up. The user will be prompted with a list of
commands:
        'put <filename>' - will upload a file
        'get <filename>' - will get a file
        'ls' - lists the files on the FTP server
        'exit' - quit the FTP client
    From there the user will be allowed to type in any of the commands.
```

Inserting patterns into the ids signatures file:
There are two ways of doing this. One is to directly edit the ids signatures file yourself. This means putting in the form <name-size (two charaters)>||<pattern-size(two characters)>| where pattern is arbitrary bytes up to 32 and the sizes match. Otherwise you can use the pattern loader. The pattern loader is designed to facilitate the loading of hexadecimal patterns. The user provides the file name, signature name, and signature contents in hexadecimal.
python3 pattern_loader <ids-signature-file> <signature-name> <signature-content>
example:
python3 pattern_loader my_ids_sigs.txt newname deadbeef

Project structure:

Server-side:
The server side is divided into 5 files with 3 main code files, one error function, and one header file. At a high level, server_handler.c contains code to set up a listening port on the machine and to wait for connections, passing them to the ids in a function call and handling sequential connections if need be. It also handles input checking and parsing of the ids_signatures file. The ids.c file contains functions relating to the operation of the ids. That is it contains a primary ids function that handles receiving and sending to the client. This during the receiving and sending portions of this function the function scan data is called on successive blocks of input. ScanData pattern matches the block with the ids signatures. After read completes whatever wasn't dropped and logged by the ids is placed in a buffer and passed to the ftp part of the application. The ftp.c file contains this part. While transfer between the two occurs via a function call passing a complete message buffer (less dropped packets) the ids itself checks data in increments of CHUNK which is defined in the header file (used to approximate the MTU of a regular network in accordance with piazza post cid=53). The ftp.c file itself contains a primary. The primary function contains the logic to choose which command was sent.

Client to FTP server message structure:
Network-format 32 bit integer representing total size followed by a char containing 'G', 'P', 'L', or 'E' followed by a space character representing get, put, ls, and exit commands respectively. This is followed immediately by the content of the message. For a get command the content is the filename. For a put command the content is a string representing the file name (must be in string format) followed immediately by the content of the file. For ls and exit there is no more content after the command.

FTP server to client message structure:
Network-format 32 bit integer representing total size followed by byte string containing content of the response. For a get this is either the file content or a message reading "Couldn't open the file". For a put this is a response either confirming writing the file or reporting failure to write file. For ls it is the content of the ls command executed by the server. There is no response for an exit.
utils.h
typedef struct transport:
This struct contains a size and message and is used for containing data byte strings that do not have a defined end marker. Input and output handling have to be able to accommodate arbitrary byte strings so a constant end marker is unfeasible. Additionally when parsing the names and patterns for the ids we have to again be able to handle arbitrary byte strings.
ErrorOut: Function to print and exit on error.
FTPExecute: Primary function of FTP server (ftp.c).
IDSHandler: Primary function of IDS (ids.c).

server_handler.c

ids.c
void IDSHandler(int client_socket, transport ids_signatures[], char * ftp_dir, char * ids_logname, char *ip):
Function Intent:
This function contains an logic for handling a connection and filtering data using functions below.
Function Structure:
Commands are handled in a synchronous fashion, that is a command is received and a response is sent before the next command is processed, the client is expected to do the same. The main body of the function uses an outer loop where each cycle should occur with one command input and output. This loop continues until the client dies or submits the exit command. Within this loop

there are two other sections containing loops. The first, the Input Section, contains a while loop that receives blocks of information from the socket at the maximum size defined by CHUNK, checks whether they contain any disallowed patterns and places the allowed ones into a buffer. Disallowed packets are logged. This buffer along with a maintained size over allowed packets are passed in a transport to the FTP server via a call to FTPExecute. FTPExecute returns a transport that contains its response and a size. This is handled by the output loop. Since the buffer is present initially and the size the client is sent has to match up with the size the client receives we utilize a for loop over the FTP's returned buffer. This for loop parses the message in blocks of maximum CHUNK and uses ScanData to check them for patterns. Those that have no matches are passed into another buffer for sending to the client. Those that are disallowed are logged and ignored (ie not copied to the output buffer).
Params:
int client_socket: int representing the socket to operate on.
transport ids_signatures[]:
array of transport objects. Even transports contain ids and odd transports contain patterns. Each id-pattern pair is stored adjacently in (0-1), (1-2) and so on.
char *ftp_dir: this variable is a pointer to a string containing the user inputted directory for the ftp server to use. Can be relative or absolute.!!!!!!!!!!!!!!!!! is this true.
char *ids_logname: File that the ids should use to write logs to. Must be local.
char *ip: The ip of the connection is passed in as a string for the purposes of logging on dropped packets.
char *ScanData(char *data, int length, transport signatures[]):
Function Intent:
This function is intended to handle scanning a block of data for patterns defined by the user. Scanning is done in a loop over the array of transport structs containing signatures and uses the function memmem to check for matches.
Returns:
The id of the first match or an empty string.
Params:
char *data: bytestring containing data to be scanned.
int length: length of said data.
transport signatures[]: array of signatures in the above described formatted used to do pattern matching. Note pattern matching is done using gnu standard function memmem which works like strstr but without the string formatting requirements.
void WriteToLog(char *ids_logname, char *id, char*ip):
Function Intent:
This function is intended to carry out logging for the ids. It takes a log filename an id and an ip and writes a log line containing the id, the ip, and a timestamp.
Params:
char *ids_logname: file to write to.
char *id: id of the pattern matched in the file.
char *ip: ip of client in connection that generated dropped packet.

ftp.c
transport FTPExecute(transport input, char *ftp_dir):
Function Intent:
This function is intended to take transport containing a message and length from the client, execute the contained command and return a response transport for sending to the client.
Function Structure:
The function receives a transport struct containing a message. This message should as its first byte contain a 'G', 'P', 'L', or 'E' representing the command type. This is used in a switch statement to direct to the different commands.

'G' aka 'get' command:
The filename is parsed out of the message starting at index 2 and appended to the ftp_dir path provided to the function. This file is then opened. If the open fails then a failure message is provided to the client. If succeeds the file is placed in a transport struct for returning.
'P' aka 'put' command:
Filename is again parsed out and appended and opened. Failure is checked. Otherwise requested file is placed in the transport struct.
'L' aka 'ls' command:
ls command is put together and executed in a pipe (popen). This pipe is read and its content is returned to the client. In the case that the command fails a failure message is returned to the user.
'E' aka 'exit' command:
No response is provided. Exit() is called.
Returns:
A transport struct containing a response message and length.
Params:
transport input: this transport contains the message to be processed by the ftp (containing command and data information) in addition to the size of the message after modification.
char *ftp_dir: the directory to read and write files on gets and puts.
ErrorOut.c
void ErrorOut(char *msg):
Function Intent:
A error printer exit helper.
Params:
char *msg: error message string to be printed.

Client-side:
On the client side there is only the file client.c. This file is largely similar in structure to ftp.c with the addition of connection handling done in file and distribution of code into helper functions.

client.c
int main(int argc, char *argv[]):
Function Intent:

```
    Function Structure:

    Returns:

    Params:

int cmd_helper(char* cmd):
    Function Intent:
        A function to read the first portion of a message and return an integer
for use in a switch statement for the different commands.
    Function Structure:
        A series of if statements using strncmp map between the input command
cmd and the returned integer.
    Returns:
        An integer from -1 to 4 representing the command.
        0=put
        1=get
        2=ls
        3=exit
```

```
        -1=other/unrecognized
    Params:
        char *cmd: a string to be compared as a command label.

int get_fname(char* cmd, char** fname):
    Function Intent:
        given a command and a filename string pointer pointer. This function is
used to check if the string given in command is not a directory. Also puts it
into a fname string.
    Returns:
        -1 if fname contains '/'
        1 otherwise
    Params:
        char *cmd: a string containing the filename starting at index 4.
        char **fname: A char pointer pointer pointing to a char * that the
filename will be placed into.
int get_path(char* path, char* fname, char** totalpath, int check_exists):
    Function Intent:
        A function that concatenates a path and filename and checks if it
exists for use in put command lookups.
    Function Structure:
        A series of strncpys to move data into the totalpath variable and then
use of stat and S_ISDIR to validate complete filepath.
    Returns:
        -1 on invalid filepath
        1 otherwise
    Params:
        char *path: path string.
        char *fname: filename string.
        char **totalpath: the concatenated totalpath (path and filename) of the
file.
        int check_exists: int (used as boolean) to enable validation.
Validation is not needed in the case of concatenation for gets as the fname has
been previously validated in get_fname.

void recv_response(int* sock, char** response_buffer, int* rec_len):
    Function Intent:
        A function wrapper for response waiting and reception after sending a
command.
    Function Structure:
        The function takes a socket, a pointer to a response buffer and a
pointer to a length integer and receives from the socket until it has received
rec_len bytes. These are placed in the response_buffer.  This is done in a
while loop reading in chunks from the server.
    Params:
        int *sock: integer representing socket to communicate on
        char **response_buffer: pointer to a buffer (assumed to be of at least
rec_len length) to place data into.
        int *rec_len: a pointer to an integer containing how much data should
be received.
```