Lab4

Name: Kwai Hung Shea

UID:304497354

Name: Fnu Pramono

UID:604498984

Q1 and Q2

According to the spec, in certain condition there will occur a buffer overrun bug. We found out that whenever we use strcpy, to cpy the filenames for downloading/uploading, the buffer overrun bug will crash the client. We changed strcpy() to the safer strncpy(), and used strnlen() to get the filename. In the case of downloading request, compared it to FILENAMESIZ, and also appending a nullbyte at either the filename length's or at index FILENAMESIZ if the length is longer. We also found other vulnerabilities in the program.

We need to create a limit maximum number of concurrent running upload process, otherwise DDOS/infinite forking may occur. DDOS occurs when a peer repeatedly asks us to upload, and since we fork each time, it can crash our system. To prevent this problem from happening, To prevent this, we created a limit to the maximum number of concurrent running upload processes of 8. Whenever we fork, we would increment this counter by 1, and we don't fork unless the counter is less than 8. When it is more than 8, we will wait until 1 upload process finishes before forking another.

Inside the osppeer.c we define a global variable #define MAXNUMUPLOADSALLOW 8

We also need to add few lines inside the original while loop to prevent the number of upload file more than the limit which is 8.

```
// Then accept connections from other peers and upload files to them!
        while ((t = task_listen(listen_task)))
        {
                if(numUploads > MAXNUMUPLOADSALLOWED) { //Infinite fork
prevention, if we've reached max allowed
                                                        //number of uploading
processes, wait until 1 finishes.
                        waitpid(-1, &status, 0);
                        numUploads--;
                }
                else
                        numUploads++;
                pid_t newPid = fork();
                if(newPid==0)
                {
                        task_upload(t);
                        exit(0);
                }
        }
        return 0;
```

The another vulnerability we found in the program is that there is no limit on the file size we download. The attacker upload an infinite amount of bytes to the downloader potentially flooding their disk space and corrupting other data. To prevent infinite sized files when we download, we added a hard limit of 1MB for any file we download. The file size limit should actually be kept by the tracker, but for this lab the tracker doesn't keep track of the size of files, so we're using a hard limit of 1MB. Once we've written more than this amount to disk, we will quit the downloading process and attempt to delete the downloaded file. We chose 1MB since the cat images are only 40KB large, so 1MB is plenty large.

In the osppeer.c we add the follwing few lines to check the download size limit.

```
// Check if we've reached the maximum allowed file size to be downloaded
               if (t->total_written > MAXFILESIZ) {
                       message("* Download file is larger than MAXFILESIZ
allowed of: '%d' bytes and will be removed\n",
                                               MAXFILESIZ);
                       message("attempting to remove file: '%s'\n", t-
>disk_filename);
                       if(!remove(t->disk_filename))
                               message("file deleted successfully\n");
                       else
                               message("unable to remove file\n");
                       task_free(t);
                       return;
               }
```

The another vulnerability we found in the program. When serving a peer, peers/attackers could try to access outside the directory and get other files. To limit =-where peers can download files from us, we implemented a check to look for any '/' characters, which are not allowed as file names in Linux. We also checked if the first character is a nullbyte, which indicates an empty file name, which may cause errors. In both case, once dectected, we goto exit and remove the upload task. We add the following if condition for detection.

```
if(t->filename[i]=='/' || t->filename[0] == '\0')
               {
                       error("* Peer attempted to access file not in current
directory %s (will now exit)", t->filename);
                       goto exit;
               }
```

Q3. The first attack is a buffer overflow attack aimed toward a peer when we request a download from them. What we do is, when we use the oosp2p_writef() to connect to the peer, instead of asking for the proper file name stored in t->filename, we ask for some random name of 5000 bytes long. If the peer does not properly check for our request, they will attempt to copy 5000 bytes to their buffers and thus overflowing their buffers. We also added an line that if uncommented would try to download "./answers.txt" from the peer's computer

The second attack we created is to upload an infinite amount of bytes to the downloades ,potentially flooding their disk space and corrupting other data. We try to fill up the victim's hard drive (as well as causing it to indefinitely hang if it processes requests sequentially) by uploading the contents of /dev/null or /dev/urandom. Thus we chew up the victim's CPU and memory resources.

The Download Attacks.

We attempt to access "/dev/zero" as to get the victim stuck in writing indefinitely and using up it's CPU cycles. Our attack uses a path with many instances of "../" followed by "/dev/zero" in an attempt to traverse the directories until we hit the root. Next we append "/dev/zero" with ".../...//" for our attack file path. Thus a single pass to remove instances of the string "../" while keeping the rest will in fact leave behind another collection of "../" repetitions. By doing this people will the caught off guard if trying to sanitize their input path with a single pass.
Next we specify a path of many repetitive characters meant to overflow the file name buffer, corrupting the victim's stack. Alternatively, we try the same attack but with many NULL bytes instead. Finally, we continuously send file requests to the victim (up to 1000 times), while alternating our previously described attacks in an attempt to monopolize the victim's resources if not crash it entirely.