## Design Project Lab 4: Authentication & Authorization

## 1) Security Objectives:

We are tackling the problem of authenticating and authorizing users to access content that the content owner considers sensitive. Before we go into more depth, it is important to note the distinction between authentication and authorization. Authentication is the verification of an entity’s identity when it is requesting access to something, such as a file. Authorization is the actual specification of the entity’s access privileges to something protected based on its identification. These concepts are central to an effective access control model.

Authentication is fundamental to security and has therefore been largely researched. Networks, including the internet, have numerous authentication protocols to ensure the packets they are receiving are from expected or otherwise trusted sources and that the contents have not been interfered with. Protocols such as HTTPS, SSL, and SSH all implement authentication procedures to verify users.

Authorization is usually the step undertaken after authentication. An entity might authorize another entity to use some of its services, such as a server allowing an authenticated entity to access certain files. Authorization is often broken down into two stages: a “definition phase” for putting an entity’s access rights into writing and an “enforcement phase” where that entity is checked before those access rights are actually handed over. For our project, we utilized Apache’s documentation on both Authentication and Authorization for our security design.

**Implementing Password Files**

htpasswd -c /usr/local/apache/passwd/passwords <user name>

This command creates a password file. In addition, we will have to create server configuration files or directives that allow us to do this. These directives can be placed in an .htaccess file in the respective directory  
  
AuthType Basic

AuthName "Restricted Files"

# (Following line optional)

AuthBasicProvider file

AuthUserFile /usr/local/apache/passwd/passwords

Require user <user name>

The AuthType is Basic (referring to mod\_auth\_basic) and will be used to authenticate the user. This method should be used in conjunction with Secure Sockets Layer (mod\_ssl), since it otherwise submits data to the server unencrypted.

**Potential Problems**

Since we are dealing with access control in the context of a peer-to-peer (P2P) system, there are some complications that must be handled differently than if we had a standard client-server system. For example, a client-server model consists of a central server that can be easily configured to protect files and control permissions, whereas each peer in a P2P model acts as its own “server.” If we use Apache’s .htaccess files for access control, they would have to be contained in the download directory of every single peer that is uploading files to the network. These .htaccess files would act much like those within a client-server architecture, but every time a peer downloads some file, the peer also downloads the .htaccess file.

Another problem is that Apache’s htpasswd function creates a password file that is in plain text. Apache offers an option to use a database to store the password file. However, in a P2P setting, a database to store the passwords might not be a good choice and rather defeats the purpose of having peers.

Furthermore, if a trusted peer downloads files from another peer, it can then choose to upload files to another peer. However, how would the trusted peer know which other peers are trusted? It would need access to the .htaccess file as well as the password file. Now there are two design options that can be done to address this. Every trusted peer can contain the .htaccess file and the password file; however, this is a security vulnerability, since many peers will contain the password file. Even if it is encrypted, it’s still a security issue. Another option would be to store the password file on a trusted server that every peer refers to see if another peer is trusted. For example, if another peer wants to download a file from a trusted peer, it will submit its username and password to the trusted peer, the trusted peer will then check its .htaccess file to determine if requesting peer is from a trusted domain, and then send its credentials to the server to verify. Once the server verifies its credentials, it will then allow the peer to download the files.

Lastly, as an issue that plagues all networks is that a malicious user can masquerade as a legitimate peer and either download files he/she shouldn’t have access to or upload harmful files to the requesting peer. It can even use the requesting peer’s credentials as its own if it is not submitted in an encrypted format. While many of these concerns are extremely difficult to prevent and out of the scope of our project, we will do our best to design a system that can mitigate damages and address these problems in the future

## 2) Overall Design:

Earlier, we introduced the motivation behind implementing authentication and authorization for a P2P model and presented the foundational basis (that is, Apache’s .htaccess and its associated files) on which we will design and build our access control. We also addressed some of the major shortcomings that arise from establishing security in P2P. Now we will outline the actual high-level design.

1. **Creating the .htaccess file**

In Section 1, we gave an example that demonstrates how to create a password file that is associated with its .htaccess file. Below is a more in-depth sample of what the .htaccess file should contain for stricter access control.

AuthType Basic

AuthName "Restricted Files"

AuthBasicProvider file

AuthUserFile /peer/local/apache/passwd/passwords

AuthGroupFile /peer/local/apache/passwd/groups

Order allow,deny

Allow from <domain, IP, … >

Satisfy all

Require group <GroupName>

Again, we are using Basic AuthType with SSL to encrypt transmitted data. In addition to using the password file, a user group file can be created to place specific users into a group. This can make the actual specification of permissions in authorization a bit simpler when we allow or disallow permissions to an entire group, such as Require group mygroup.

The Order allow,deny clause tells the .htaccess file whether to allow or deny by default; the second argument is taken by default. In this case, we are specifying which users or groups to allow access into our network while denying entry to all other unspecified peers.

In addition to users and groups, Allow from <domain, IP, … > allows certain IPs to gain entry into our network. For stricter control, Satisfy all means that all allow clauses must be satisfied. In our case, the user must have the correct password, be in a user group with adequate permissions, and connect to the network using an appropriate IP address or domain.

1. **Filling in the password and/or user group files**

Based on how we utilize .htaccess to allow or deny access to our files, we may need to set up additional files to act as an Access Control List (ACL). As mentioned earlier, the password file specifies user-password combinations that are meant to have permissions to download files, while the group file places users within groups for simplified permissions.

The password file follows a format much like:

username:bAWrXFTOqBZps

nathant:Re.qPW36O46MM

arunj:Dh0gXzqi.M1S2

username:password

**IF ENCRYPTED**

nathant:youshallnotpass

arunj:p455w0rd

The group file follows a similar format:

mygroup: arunj nathant

1. **Implement parser and checks in osppeer.c**

With the actual definition phase of authorization done, we can move onto the enforcement phase. When a peer is downloading some file, the actual code in the uploader’s osppeer.c will check if the (uploader’s) .htaccess file exists. If it does not exist, the uploader most likely does not have any sensitive files to protect, and we can proceed as normal without authentication or authorization.

But if the .htaccess file exists, we need to parse it. Immediately prior to parsing, the file syntax is checked. Improper, erroneous syntax that makes parsing impossible will stop file sharing from the uploader, and the downloader will have to find a new peer to download from. The fact that the .htaccess file exists means the uploader most likely meant to restrict access to his/her files, but did not do so in a correct manner. In that case, we need prevent sharing of those files and notify the uploader.

If parsing is done successfully, the code will now have variables that contain allowed IPs, user groups, passwords. We also determine the combination of these that have to be satisfied for download permissions to be granted, and whether the uploader wants to allow or deny by default.

Then we can check all these variables on the uploading side and verify that the transaction can be made. The uploader peer will either start uploading the file or return an error message to the downloader peer.

**Alternatives**

One different approach would have been to design our own access file independent from Apache. This would allow us a greater amount of flexibility in implementing a concrete security design with built-in encryption, group files, custom data structures, and checksums – all of which are catered to our specific P2P model. The downside is that we would have to design the access file from scratch rather than piggybacking off of Apache’s well-defined model. More importantly, however, the peers using the from-scratch design would have to learn an entirely new syntax and process to set up authentication and authorization for their P2P network.

Yet another approach would involve creating a central server for each P2P network that stores passwords and group files for each and every peer. This central repository would receive a request for every time an uploading peer requests authentication and authorization for a downloading peer. This increases security by preventing the need for peers to pass along sensitive information at the cost of centralizing a P2P network, which is highly counterintuitive.

## 3) Security Principles

### Principle of Economy of Mechanism

In terms of the user interface, there is almost no change for the end-user peers. The only difference is that he/she has the option of adding another file that controls access to shared files. Even the parsing and checks to authenticate and authorize are internal to the code. The downloader sees a notification that either the file is downloaded, or access was not granted. The uploader sees a notification that either their file was shared, or permission was denied to another peer.

### Principle of Fail-safe Defaults

In general, if a peer decides to use the .htaccess authentication and authorization implementation we designed, any error should fail safely. For example, if the uploader defines a .htaccess file with incorrect syntax or some logic problem that causes the parser to fail, we would prevent the download from taking place completely. In other words, as long as the uploader creates a .htaccess file, we assume he/she wants to protect their files. And if the .htaccess file is empty or does not make sense, we default to a deny-all option while alerting the uploader.

### Principle of Open Design

As a matter of fact, peers can easily obtain other peers’ .htaccess files, including the passwords. However, if the protective peer follows guidelines on encrypting their passwords, other peers should not be able to do too much damage.

### Principle of Separation of Privilege

Like Apache, our design allows the uploader to protect his content with n-factor authentication. For example, 3-factor authentication might require the user to type in a password, be in an acceptable user group with express download permissions, and also be logging in from a valid IP or domain.

### Principle of Least Common Mechanism

Unfortunately, due to the P2P model, peers necessarily have to download .htaccess and password files from each other in order to authenticate and authorize users. However, encryption can make these files a bit more secure.

### Principle of Psychological Acceptability

Peers can easily choose to add a layer of protection to safeguard their files from other unwanted users. The only thing that has to be done is to create a .htaccess file with the permissions listed (and possibly password and group files if necessary). If the peer does not create a .htaccess file, then the program will not have anything to parse; in this case, the user has opted out of using control access.

## 4) Challenges & Conclusion:

We faced major challenges in implementing the interpreter that would parse .htaccess files in order to store the uploader’s security directives into a corresponding data structure. We also had issues with formatting the downloading peer’s request to the uploader and properly encrypting the request.

One considerable flaw in our access control is the overhead involved in passing .htaccess, password, and group files between peers. If the files are small in size, little performance cost is incurred. However, if the end-user unwisely chooses to deny on default but specifies millions of user:password combinations to allow, not only does that huge file have to be downloaded, but our interpreter will have to traverse through each and every user in the list before granting or denying download permissions. Since we are using a P2P network and not a central server, there is little we can do to reduce this overhead unless we strategically place a size and line count limit on the access files.

To conclude, we believe our design is a formidable intersection between practicality of use and actual security. While a good design should be secure, it should at the same time not compromise on ease of use.

## Work Breakdown:

Presentation Slides:

We both worked on slides together and presented alternatingly. No demo was done in class.

Documentation:

Most of the research and early stages of writing was done independently. We then met up and consolidated both research knowledge and documentation progress.

Implementation:

Pair-programming was utilized in the making of our authentication/authorization design.