FINAL REPORT

CLIENT/SERVER AUTHENTICATION

For: COMP3203 Principles of Computer Networks

Michel Barbeau

By: Jacob Brown

Student Id: 100 762 690

Due:

**Title**: Client/Server Authentication

**Date**: 5 December 2014

**Authors**: Jacob Brown – Carleton University – 100 762 690

**Note**:

The work in this report is new and was prepared the course COMP 3203 Principles of Computer Networks, Fall 2014, School of Computer Science, Carleton University.

**INDEX:**

**Section 1: Executive Summary**

**Section 2: Introduction**

**2.1 - Context/Background**

**2.2 - Definition of the Problem**

**2.3 - Summary of the Result**

**2.4 - Outline of the Report**

**Section 3: Detailed Context/Background of the Protocol**

**3.1 - Other Methods of authentication:**

**3.2 - A description of Kerberos**

**3.3 - The Authentication Protocol (Like Kerberos)**

**Section 4: The Implementation and findings**

**4.1 - Features and process of designing the implementation**

**- 4.1.1 - Design Phase while planning the implementation**

**4.2 - Specifics of the implementation and Protocol**

**4.3 - Example Execution of Implementation**

**4.4 - Testing methods and findings**

**Section 5: Conclusions and Future expansion**

**Section 6: References**

**Section 7: Annexes**

**7.1 - Annex 1: Log File**

**7.2 - Annex 2: Screen shot of server with failed attempts**

**1. Executive Summary:**

The goal of this project is to demonstrate the use and importance of network authentication and how it works to provide security in a client/server environment. This project will provide details of the known problems and vulnerabilities in many computer networks, specifically networks connected to the Internet. Current authentication methods that have been used in the past will also be analyzed as a tool for reasoning why the method of authentication used in this project was chosen.

**2. Introduction:**

The Internet is a global network system that is interconnected between many other computer networks that use the **standard Internet Protocol** or **TCP/IP** to link between several billion devices throughout the globe. With well over 2 billion Internet users on this planet sharing information between systems can be tricky. This sharing and communication between two or more systems can be especially tricky when information is being passed back and forth that is not wanted to be shared with other systems or users. This raises a security problem in the sense of how do we protect our information from not having just any user connect to our system and having complete access. While there are many methods created and attempted throughout the life of the Internet thus far, we are going to focus on the aspect of utilizing client/server authentication.

**2.1 Context/Background:**

In todays world, the majority of thing we as people do with our everyday lives, in some way incorporates the use of the Internet. The use of Internet and web based applications has grown drastically over the past decade and will only continue to grow, as seen in figure 1. Statistics gather from citation found in section 6 [1].

Figure 1: Growth in Internet users globally for the past decade.

As the growing use of the Internet and web based applications continues grow so do the security risks that come along with it. Whenever a website or an application is being accessed by a user they, knowing or not, must be verified by either the application or the server at which is hosting the page or application. This type of verification can be as simple as confirming a request was sent from a client to access a page to encrypting information over a network to verify secret credentials. At every moment an unimaginable amount of information is being passed over the internet with passwords, bank account numbers, credit card number, social insurance numbers, and much more. This is why security is so important. With all this information out there being passed from one machine to another how can we know if someone really is who they say they are when they are requesting or giving information? How can this information be prevented from being stolen?

**2.2 Definition of the problem:**

This project is to provide an implementation of an authentication protocol that works behind the scenes of a client and server environment to authenticate a client to a server without ever passing any valuable information across the network. This is known as a zero-knowledge proof. This means that when a client is asked to sign in and authenticate themselves to the servers, they do not actually send the password to the server, but rather prove somehow that they do in fact know what the password is. This protocol should also give some sort of security against possible attacks from high-jacked accounts as well as preventing too many attempts to authenticate from a user with incorrect credentials.

This protocol should not interfere anyway with performance of the server or client, and should act as if it isn’t even there when clients are asked to authenticate themself. That is, should be seamless in operation and not visible to the user.

**2.3 Summary of the result:**

The protocol developed is similar to that of Kerberos in the sense that instead of sending a password from a client to a server we rather send a hashed value that represents that of the password. By doing this no valuable information is being sent over the network where an attacker could gain access to such information. In the protocol developed a client sends a request to the server in the form of a username. The server has a list of usernames and passwords and retrieves the appropriate password corresponding to the username. The client then runs a, protocol developed, hash on the password. The has function produces a key and salt value to hash along with the password and returns the result to the server, along with the generated salt and key. The salt value and the key is sent to the client, where they will repeat the same steps but provide the key and salt this time. Upon completion the resulting hash is sent back to the server to be compared to the hash generated on the server. If the two hash values match we authenticate the client.

**2.4 Outline of the report:**

Section 3 gives a detailed context/background on the authentication protocol developed and explains how it solves several problems that are addressed in many real world problems today. Section 4 gives a detailed look into the implementation of this protocol and how it compares to that of Kerberos. Section 5 will address the conclusions gathered during the development of this protocol and its implementation as well as room to grow in the future.

**3. Detailed Context/Background of the Protocol:**

Authentication is the process of proving to someone or something that you are who you say you are. A good analogy used is when a police officer asks you for Id so they can tell who you are. When the officer looks at the picture on the card and then looks at your face he compares the two. If you match the picture on the card then he agrees that you are who you are supposed to be. This process is the same for authentication. There are several ways that someone or something can be authenticated. The first part of this section will briefly touch on other methods used for authentication and where they are used, addressing some issues that come along with those methods. The second section will briefly describe what Kerberos is and how it over comes such issues. The final section will explain in detail how the authentication protocol in this report addresses those issues as well and how it overcomes them.

**3.1 Other Methods of authentication:**

One method that can be used for authenticating a user is the uses of encrypted key exchange, which is the method of passing the key over a network in encrypted form. This raises an issue of if someone is eavesdropping, they can obtain the encrypted password and possibly break the encryption and gain access to the password with out the originator ever knowing. There is also ticket-granting systems which hand out tickets, which are files, to users that authenticate that user for a given amount of time until the ticket expires. A possible issue with this is that, if an attacker obtains one of these tickets before the time of expiry, then the attack has access to the system until the expiry date. Each one of these methods on its own appears to be good but still brings forth some very big security risks that can be prevented with some modification.

**3.2 A description of Kerberos:**

The Kerberos protocol, developed by MIT, has come to be a standard in todays Internet protocols. This protocol also works with a ticket-based system over a non – secure network to authenticate one node to another. Nodes being in respect to this report a client and/or a server. Kerberos overcomes the issues in section 3.1 by incorporating several features. These features include a session key, ticket granting ticket, and time stamps. How it works is a user sends a request to an authentication server in plaintext made up of the user/client username. The authentication server then looks up the password for that username and generates a hash value based on that hash. The server then sends back a session key that is encrypted used the secret key of the user/client, as well as the ticket generated by the ticket granting system. The ticket is encrypted using the session key that was encrypted by the secret key of the user/client. Once the client has received the two items, it then attempts to decrypt the session key using the user inputted secret key. The session key is what will be used along side the ticket, which is time stamped, to authenticate itself to the system. Note: if the client’s password is not correct the decryption of the session key will fail and the user/client will be unable to authenticate [2].

**3.3 The Authentication Protocol (Like Kerberos):**

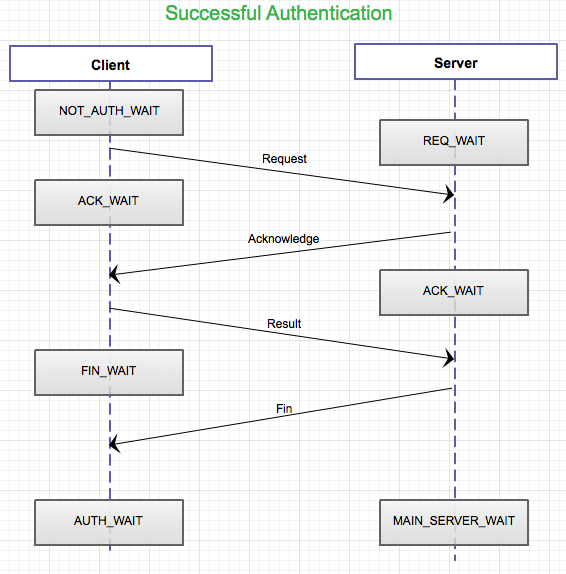
The protocol developed for this report has similar features to Kerberos and as a result has been labeled as a “like Kerberos” protocol. Using similar features as Kerberos, this protocol neglects to use a ticket granting system. The features this protocol does use are one time salt values, public keys, and single use hashes. This protocol works as follows: A client/user submits a request to the authentication server in the form of a username. The server looks up the username and retrieves the appropriate password. The authentication server will then perform its own hash function on the password that generates a one time salt and one time key to be hashed alongside with the password to create a single hash valid for one login attempt. This prevents the ability for an attacker to obtain an old hash and use that value to gain access to the server. After the server has produced this hash value, it then sends to the client the produced single use key, and single use salt. Upon the client receiving these two values, it then runs the hash function on its side with the user-inputted password, providing the single use key and salt value.

Now that the server and client both have a hashed representation of what the passwords are believed to be they must be compared. The client then sends the result back to the server for the comparison. If the values do not match, then the user-inputted password was incorrect and is not authenticated. The salt and key also are no longer valid for farther attempts.

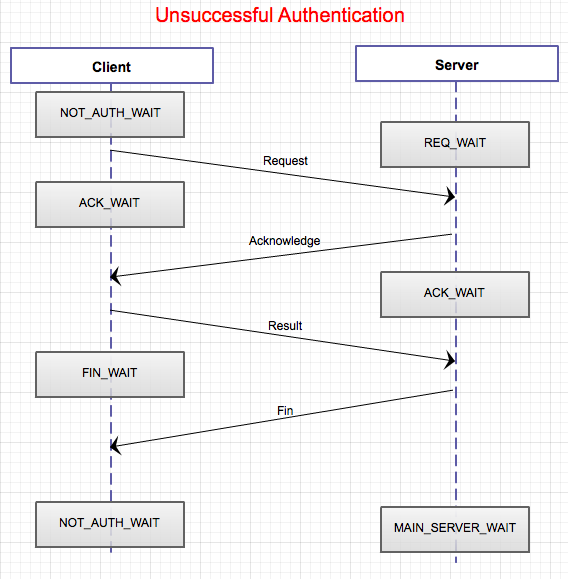
If the client/user is authenticated, meaning the hashed matched, they are then added to a variable holding authenticated users on the authentication server and they will have access to that server until they logout.

If a client/user fails to be authenticated, and attempts again, the number of tries will be recorded on the server. If the client/user attempts 3 times, they are then locked out of the authentication server until released by an authenticated user. This action will also be logged and recorded as a possible security threat of a high-jacked account. Once an authenticated user has released the locked user/client then they can re-attempt to log in and are given another 3 tries.

By incorporating the above features we have addressed the issues in section 3.1 and have shown the similarities of that of Kerberos, mentioned in section 3.2.



*Figure 2: Shows a successful connection to the server, following the steps explained above.*



*Figure 3: Sequence diagram of unsuccessful authentication. Note “NOT\_AUTH\_WAIT” indicating waiting for Client to request Authentication again.*

**4. The Implementation and findings:**

This section will describe the process of developing the implementation and the features at which the implementation has. The reasons as for the type of server and language chosen to program in will also be discussed. Several testing methods used to determine usability, effectiveness, and reliability are also mentioned at the end of this section.

**4.1 Features and process of designing the implementation:**

***4.1.1 Design Phase while planning the implementation:***

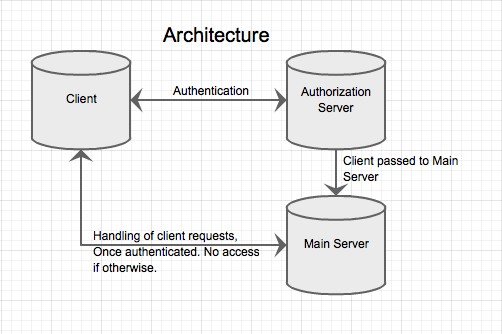
*The implementation of this client/server authentication will consist of some sort of hashing algorithm used to determine if a user is valid or not. Since we do not want to pass any passwords over the network, as this is not safe, we want to send a hash back and forth instead that will mean nothing to an attacker if intercepted. Of course there are other issues with this, but they will be addressed in the full report of the project. How this will work is the server will store an encrypted list of passwords in relation to users. Both the client and the server will have a common hash function, and the server will generate single-use “salt” value. What happens is the server will generate some random salt value and send it to the client, encrypted. The client will then concatenate or xor this value with the password, and run the common hash function on it (Hash (password, salt value)). The client will then send this value to the server, which has done the same and compare the two. If they match, the client is then authenticated. The reason for single use salt values is because there is a possibility that an attacker may have intercepted the previous salt value and if a new value was not created upon failure, could guess and possibly gain access to the system. Note, this is a very similar implementation of the Kerberos protocol developed by MIT.* [*http://web.mit.edu/kerberos/*](http://web.mit.edu/kerberos/)

As planned, the above is how the protocol operates. However as the implementation came together more was needed to be added to make the implementation more robust, effective, and seamless to the client/user.

The first feature was implementing a variable on the authentication server to keep track if the client/user attempting to authenticate is in fact authenticated or not. As well as the use of this variable we also must keep track of the number of attempts a client/user sends to the server to prevent an attacker from sending countless requests for a given username and brute forcing their way into the server. This then required the addition of a file to store usernames of accounts that are locked out of the server to check incoming requests against to prevent them from attempting to login. A log file was also added to the server to log any possible security breaches and risks that may be raised during the operation of the server as well as any major failures, such as a missing login file. An example of the log file can be found in Annex 1. A final touch was added to the server color-coding the display of requests to make reading the server easier and recognizing security threats quickly. See Annex 2 for an example. Note all examples show information on the server not encrypted where as in a real world implementation all files would be encrypted protecting user information.

**4.2 Specifics of the implementation and Protocol:**

The implementation of this protocol including the protocol it self have been written in the standard C language. The reason for this is due the fact that c is a lower level language and provides the least security risk compared some other languages such as java. Java is heavily object based and this can result in security flaws within the language it self. C in a procedural language and does not have these issues.



*Figure 4: The architecture of the implementation.*

The server and client both use the UDP library and have been made as a UDP system. UDP was chosen for this implementation for two reasons. The first reason is because UDP does not keep an open stream between of bites between the server and client. We do not want an open stream for the reason that this raises the security risk that a malicious user may send malicious data over the open stream and infect the server. The second reason is that this is a Kerberos like implementation and Kerberos uses the UDP protocol as well for the same reason as previously mentioned.

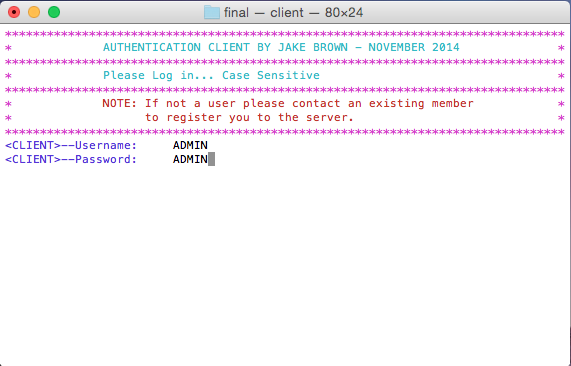
The implementation allows for one client at a time to access the server just to demonstrate the authentication protocol but with modification can be made to support multiple users through the use of a user stack. The implementation of the server has been written in a way to be built on later down the road. As for the protocol its self, it has been written strictly for the purpose of studying and researching how these protocols work and as a result has been written with this in mind using self developed hash functions and non cryptographic pseudo random number generators. The specifics of the protocol even though not made to be implemented in the real world still satisfies the needs to be effective as an authentication protocol.

One-time use keys are 32 characters long resulting in 256 bits. Modern key sizes for hash functions use keys of the same size and therefor this is a reasonable choice for the key size. Accompanied is a salt value of up to 10 integers long and no shorter than 9. When hashed together with some password provided a 32 character or 256-bit hash value is generated. Keys are generated using a simple random number generator that produced an index number the correlates to a character in a character array. This generates 32 characters randomly and returns the result as a key. The salt value is just a large random number chosen and concatenated together to create a number of appropriate lengths. Finally the hash value takes both values and the password in plaintext and does simply 3 operations on them before returning the final hashed result.

**4.3 Example Execution of Implementation:**

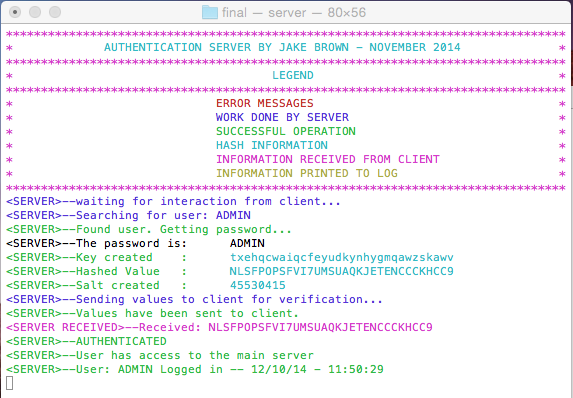
Using the provided implementation can be done so as follows:

Once the client and server are both running, the client will need to connect to the server. Note that the multi colored text in the header of the server found in figure 6 is the legend for outputted results on the server machine. In order to do so the client sends a request to the server in the form of a user name. This request is not sent until the password has been entered and the user presses the return key.

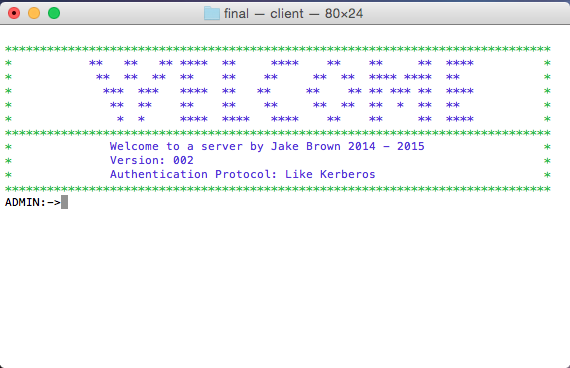


*Figure 5: Admin account being logged into on the client machine.*

Once the client has pressed the return key, the username is sent to the server and the server retrieves the password for that username located in a central database located on the server machine. The server runs a hash function, found in authentication header file, which produces a one time use key and salt value to send back to the client. The client receives the one time key and salt value to hash the user inputted password. The resulting hash value on the client side is what we send back to the server for confirmation. This is how we prove the client knows the password, without ever sending the password over the network. This is known as a zero-knowledge proof. Once the server receives the hash value, the two hash values are compared, the clients hash value and server hash value. If both are equal, then the server authenticates the user.



*Figure 6: A sample authentication of the ADMIN account. Note the two hash values. The Cyan color being the server produced hash value, and the magenta being the client generated hash value.*



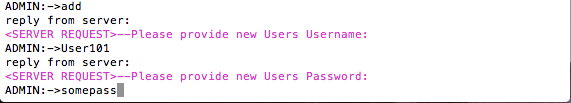
*Figure 7: The welcome screen on the client machine once authenticated.*

From this point the client has several commands they can execute. The best thing to do is type the help command. This will return a list of implemented commands on the server.

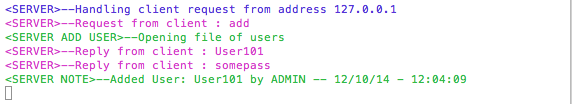


*Figure 8: The help command executed on the client machine.*

If no users are located in the server database, we must then go ahead and add a user. To do so the client must type the “add” command, at which the server will request a username and password from the client. The username and password provided is then added to the user database on the server. Figure 9 shows the client side and figure 10 shows the output on the server side.



*Figure 9: The client side answering the request for the new username and password from the server.*



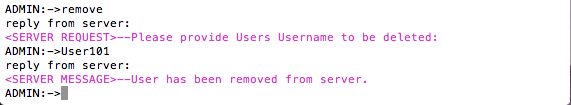
*Figure 10: The server receiving the new username and password and adding the new user to the database.*

Now that we have added a new user we can view all users in the server by simply typing the “users” command.



*Figure 11: The “user” command executed by the client.*

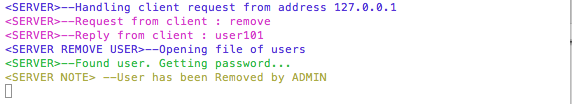
Ass easy it is to add someone to the server we can also remove someone. All we need to do is type the “remove” command and supply the username requested by the server and the user is then removed from the server.



*Figure 12: The remove command being executed on the client.*

In figure 13 we see the server side of things during the remove command. The server receives the username from the client, displayed in the magenta color. The server then opens the file of users and searches for that username, which if found, removes that username and that the corresponding password and write to the log.

The log keeps note of who removed what user and the date and time of the interaction.



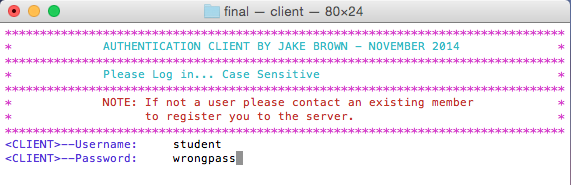
*Figure 13: The “remove” command being handled on the server.*

Both, the add and remove operations are logged to the log file, which if being viewed on a MAC, can be ran in live feed using the TAIL –F command followed by log.txt, once in the bin directory.

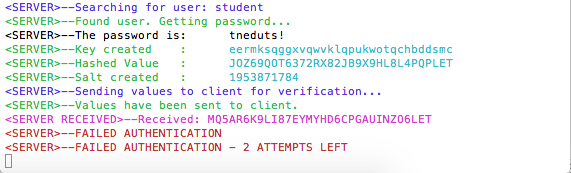


*Figure 14: Same from the log file of add and remove operations.*

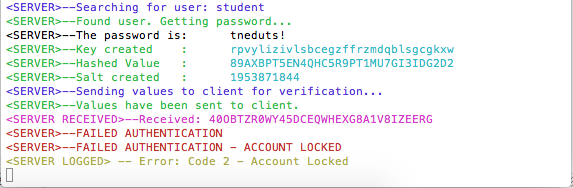
If the client attempts to log into an account but fails to provide the appropriate password, the client will be denied access to the server with a total of three allowed attempts. Upon the first failed attempt a file with the format <USERNAME>.txt is created storing the number of failed attempts. Once three attempts have been attempted the account will be locked and the user will be added to a locked users file located in the /bin directory.



*Figure 15: Client attempting to access account with wrong password.*

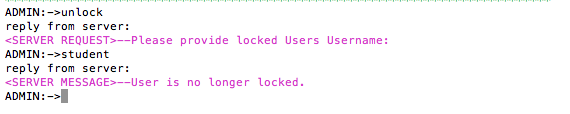


*Figure 16: Server handing the attempt denying access to the student account.*



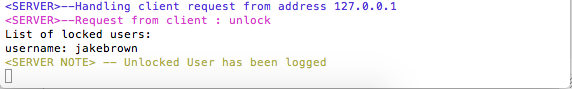
*Figure 17: The server locking the account to the user account after 3 failed attempts to log in with the incorrect password.*

In order to unlock a locked user account, the locked user must contact an authorized user to unlock their account. This is done by an account logging in as normal and using the unlock command. By typing unlock the client is asked for the username at which they wish to unlock. After they provided the correct name of an existing account that is locked they will be notified of the success of unlocking that account. We can see this process in figure 18.



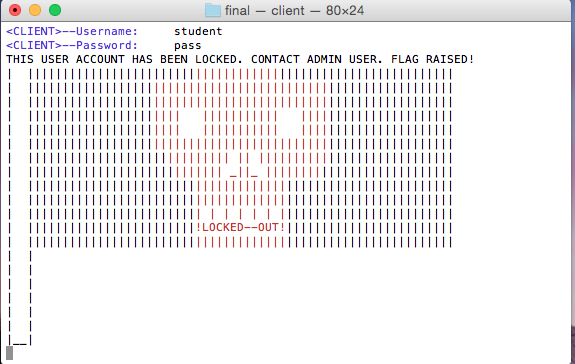
*Figure 18: Client executes the unlock command to unlock the student account.*

Similarly to removing a user, the client unlocks a given user/account and displays the result in the server window. In figure 19 we see just this done to the user account “jakebrown”.



*Figure 19: User account “jakebrown” unlocked by client and logged to the server.*

If a user attempts to log into an already locked account a flag is raised and logged to the server as a possible security risk. Figures 20 and 21 show the client receiving the flag and the log file being updated with the possible security threat.



*Figure 20: Flag raised on client machine.*



*Figure 21: Possible security risk logged.*

Now if a user were to try and access the server who is not located in the server database, this could also be a possible security risk and as a result, this action will also be logged with the attempted username with date and time. This is shown in figure 22.

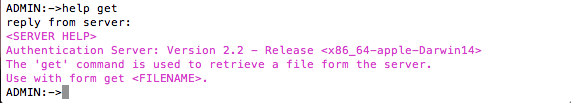


*Figure 22: Server handling a user not located in the database.*



*Figure 23: The event of unknown/not found user being logged to file.*

Some other useful tips when operating within this implementation are the use of the “help” command followed by an argument. The argument must be one of the displayed, implemented, commands provided by the server when executing the regular help command. If the user inputs an argument not recognized the command will be treated as if it were a regular “help” command. Also another key point to make is when using the “get” command to retrieve a file, using the “ls” command will return the list of files in the working directory of the server. This is where the creator of the server should place all files on the system that they wish to give access to the clients accessing the server.



*Figure 24: A sample of the “help” command being used with an argument.*

**

*Figure 25: The ls command being ran on the client machine.*

**4.4 Testing methods and findings:**

During the development of this protocol and during the development of the client/server, a significant amount of testing was needed to assure the system would not crash or get hung up during bad input or numerous attempts of authentication by a single user.

During the development of the protocol its self many small test programs were written to test the reliability of the protocol through many attempts of creating single use keys, salts, and hashes and simulation authenticating users and failures. Through this testing, some bugs were addressed and resolved. One of these bugs was the use of the strlen() function on keys to determine the length of the key when printing and passing the key in string form rather than the character arrays they were. This caused key to look past the bounds of the key itself and read the memory out of bounds. Rather than throwing a segmentation fault or causing a kernel panic, it would just return garbage. This resulted in client/server hash values being different when in fact they should be matching. This bug was over come by adding a null character at the end of the key, which tells strlen() when to stop reading. This fixed that problem.

Once the authentication protocol was working as hoped, the implementation into a client/server system was to be completed. The server was to have the majority of the code and the client just simply send requests and receive replies. This way the client has less control over the information in the server, and prevents possible attacks from a malicious user. The server was tested again possible bad data such as attempts of garbage data crashing the server. The server handles such information but sending a “unrecognized command” signal back to the client and requests the client try again. The server was also tested again multiple attempts to log in with a single username, which if not prevented, could allow for a malicious user to possibly brute force attack the server and try countless passwords till a hash worked.

One common attack against hashes is searching for a collision. A collision is when you can take some value that is not the password and generate a hash value that is the same as that created with the password. This property of hash functions is due to the fact that they take an infinite amount of data and maps it to a finite amount of data. The prevention of attacking the property is easily addressed by using the one-time use keys and salts. By the use of these values during the hashing, after the client has submitted the hash to the server, an attacker can no longer use those values to compute the hash because they will no longer be valid. Therefore trying to find a collision will be looking for a collision with an old hash that is no longer valid.

**5. Conclusions and Future expansion:**

From the testing explained in section 4.3, the conclusion can be made, that with the described protocol and implementation it is possible to overcome the addressed issues found in other protocol methods and secure a login between a client and server. This implementation also allows for a seamless login from a client to the server without hassle to the client user creating a user-friendly environment for unfamiliar users interacting with the client.

Since the protocol was developed just for the purpose of this course there are several more properties that should be added before implementing in the real world. The following are among some of these properties that should be added:

* Time stamped hash values to prevent high-jacking of hash values between client and server.
* Cryptographic pseudo random number generators for producing salt values and generating keys. This prevents an attacker from predicting the generation of future keys and salt values based of previous state of generators.

As for the implementation, there are also some improvements and room for endless growth as in the security world, there is no end to the race between attackers and security experts. There is always room to improve. Some of these improvements to be made to the implementation in the future are as follows:

* Denying access to overloaded requests from a malicious third party.
* Statistics application to monitor use of accounts and help determine if a user account has possibly been high-jacked.
* Email request password resets rather than only being able to release locked users by another authenticated user.

In final conclusion, even though the above improvements would be effective in real world use and would in fact make this protocol and implementation more secure and effective, it does however meets all the requirements it intended to for this project.

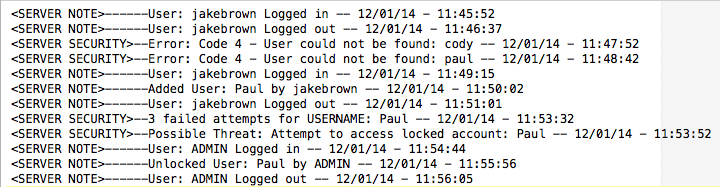
**6. References:**

[1] Stats for graph - Internet world stats - <http://www.internetworldstats.com/emarketing.htm/>

[2] Kerberos Protocol - MIT - <http://web.mit.edu/kerberos/>

**7. Annexes:**

**7.1 Annex 1 – Log File:**

****

**7.2 Annex 2 – Screen shot of server with failed attempts:**

****