Abstract

A lot of research has been carried out in Password Authenticated Key Exchange (PAKE) protocols and these protocols must remain secure even if the password is chosen from a relatively small space of possible passwords (a dictionary of English words). Most importantly, the protocol must be secure against off-line dictionary attacks. Off-line attack is a scenario whereby the attacker does not require feedback when enumerating various passwords in other to get the valid one.

Recently, Goldreich and Lindell [1] proposed a construction based on general assumptions, which is secure in the standard model with human-memorable passwords. Their protocol does not require public parameters unfortunately their protocol requires general multi-party computation which made it impractical.

Web Services are technologies that enable communication and exchange of messages between applications of diverse platform over the internet. If not properly secure, companies would not adopt it because there would be no secrecy and their intellectual property right would be bridged. Presently there exists a secure channel called Public Key Infrastructure (PKI) [2].

The Aim of this project is to investigate the efficiency of secure channel obtained through an alternative method based on PAKE. In particular, the investigation would be centred on the efficiency of the Password-AKE protocol proposed by [Jonathan Katz, Rafail Ostrovsky and Moti Yung [3]].

TABLE OF CONTENTS

Chapter 1: Introduction and Context Overview

Authentication is a paramount issue when two entities communicate over the internet. The Internet is a global system with various computers connected to it in order to get the information's they require. It uses the standard internet protocol (TCP/IP). A web service is a very important technology on the internet. It is a technology that enables communication and exchange of messages between applications of diverse platform over the internet. They aid companies and organisations in making real-time consumption/ utilisation of resources and software components outside their implementation domain. A web service is porous if not properly secured and other company information can be viewed by invalid customers. Due to this reason, researchers have been studying mechanisms that can be used to secure communication and exchange of messages over the internet. So far some secure channels have been implemented; it is based on Public Key Infrastructure examples are SSL, Kerberos. In 1994 [5], Entrust Authority created the first commercial Public Key Infrastructure (PKI). This dissertation will explain why Password Authenticated Key Exchange (PAKE) is a better method in securing communications over the internet.

1.1 Aims and Objectives

The project is aimed at investigating the efficiency of secure channel obtained through an alternative method based on Password Authenticated Key Exchange (PAKE). In particular, the investigation is centred on the efficiency of the Password-AKE protocol proposed by [Jonathan Katz, Rafail Ostrovsky and Moti Yung].

The objectives are listed as follows:

- Implement a secure channel based on Password Authenticated Key Exchange.
- To ensure that confidential messages involved in the communication between a Web Service producer and a Web Service consumer are securely transferred with significant guarantee on the integrity of the message.
- Build a prototype Web Service application that demonstrates a typical secure session achieved in the implementation of the chosen algorithm and protocol, the validity and relevance of this work to the developer community as well as to organisations.

1.2 Context Organisation

This dissertation consists of four main parts:

Firstly, the next chapter introduces the background research and related works in PKI, how it works and its components. It also gives the overview of PAKE and the specific protocol.

Secondly, chapter 3 explains the design of the project which includes the classes used, details of the technical problems encountered and tackled, and the development of the program.
Thirdly, chapter 4 talks about analysis of the results achieved during the experiment and a critical evaluation of the work.
Finally, chapter 5 gives the conclusion and some suggestions on further work which could possibly improve the performance of the project.

1.3 Introduction to Public Key Infrastructure

PKI is a technology that uses a centralised trusted third party to validate an entity during communication. The third party vouches the identity of an entity to other members. This third party is called the Certification Authority [6]. The CA provides a range of guarantees including authentication, data integrity, data confidentiality and nonrepudiation. It provides a desired level of trust using public key based cryptographic techniques to generate and manage electronic certificates. These certificates for example, X509, are used to connect an individual or entities to a public key that is used to validate the information provided by the entity/ individual or facilitate data encryption. This technique is presently used on the Internet; examples include SSL, Kerberos, and TLS.

The main component in achieving a third party in PKI is: (1) a digital certificate: links an individual to a public key (2) certification authority: creates certificates and vouches on validity of the entity relying on the PKI; (3) registration authority: responsible for verification of user's identities so that the appropriate key pairs and digital certificates can be created and (4) certification paths: recognises and trust digital certificates issued by other PKIs in order to create larger, connected networks of trust.

1.4 Introduction to Password Authenticated Key Exchange

In PAKE the user only needs to remember a password, there is no need to have knowledge of any secret or public data. Parties (two or more) can communicate based on the knowledge of the password shared among them and an unauthorised party cannot participate since it does not know the password. The purpose of PAKE is to provide a strong mutual password authentication without pre-authenticated public keys so that the only way an attacker can attack the protocol is to run a trivial online dictionary attack by simply iteratively guessing passwords and attempting to impersonate one of the parties.

Chapter 2: Background Research and Related Works

The notion of secure key exchange, symmetric cryptography also called private key system, and asymmetric key algorithm also called public key system used to secure communication, was first proposed by Diffie, Hellman, Rivest, Shamir and Adleman in the year 1976 [4]. Netscape invented the first effective PKI protocol in the year 1995 called SSL (Secure Socket Layer) [7] example is https in Web URL's. Its features include key establishment, server authentication e.t.c. TLS (Transport Layer Security) was an improved version of SSL designed to prevent eavesdropping and tampering during a client/server communication. It provides RSA security with 1024 and 2048 bit strength [10]. Digital signatures are unique to each individual. It contains information about the person and cannot be shared. It creates persistent and tamper resistant evidence of the owner. It consists of three algorithms; the **key generation** algorithm which generates the private key and public key, the **signing algorithm** which provides a signature given a message and private key, and a **verification algorithm** which provides the authenticity of a message given the public key, signature and the message.

2.1 Public Key Infrastructure Components

- Certification Authority (CA): The CA serves as the trusted third party and the verifier. It verifies the information provided by the client such as client's name and public key before they are authenticated. By signing the certificates of a client, the CA proves to other entities that it has carried out an extensive check on the information supplied by the entity and that it is satisfied with the authenticity of the client. The security of the PKI system as a whole is based on the security of the certificates. Therefore the key of the CA has to be very strong so that keys cannot be faked or forged by an adversary. Any compromise made on the CA can affect the security of the system; this is because the CA is the backbone of PKI. CA manages digital certificates. It is responsible for the generation, distribution, renewal, revocation and suspension of digital certificates.
- Digital Certificates: This is an electronic credential that contains the information that can be linked to a specific entity; it guarantees the association between a public key and a specific entity. It is created by placing the entity's name, public key and other information used to identify an individual in a small electronic document that is stored in a directory or a database. It is created by the Certification Authority.
- Registration Authority: These are bodies that tell the Certification Authority to issue digital certificates after verifying the user's request. They identify and authenticate users, they do not sign or issue digital certificates. Therefore, RA is expected to comply with the standards for verifying a person's identity.

- Certificate Policy: This is a set of rules governing the use of certificates and level of trust placed on a particular PKI. It contains items such as the responsibilities of the CA, its liabilities and warranties, confidentiality policy, identification /authentication requirements and details of what information are contained in the certificates. The certificate contains information pertaining to the subject of the certificate; this includes the version number, serial number, validity period, name and public key of the subject. It also contains the name and signature of the issuing CA.
- Certification Storage: This is a database used to store certificates. Clients make request for certificates from the certificate storage. The Certificate storage is being consulted by clients when they want to communicate securely with another client. The security lies in the signature of the certificate and not in the certificate storage.
- Certification Revocation: This is a process of revoking an expired certificate. A revocation message is sent to the CA when a certificate is to be revoked. The CA then issues a revocation notice to the Certification Revocation Server (CRS). Sometimes the database might not be updated early therefore a revoked certificate might not have been deleted on the system thereby a call for such a certificate can return an invalid certificate. There are two ways of handling revocation, these are Certification Revocation List (CRL) and Certification Revocation Notice (CRN). CRL are lists of certificate serial numbers associated with a revocation status, signed and time stamped by the issuing CA. Each revocation process generates a new list which must be sent to the CRS for storage. As time passes, these lists grow due to the revocation until they become too big to transmit. One approach to solve this problem is to split the list down to validity periods, for example monthly intervals. Delta list updates the previous lists [6]. Delta CRL is simply the list of changes that occurred since the last full CRL has been published. The delta CRLs are prone to man-in-the-middle attack. If the client does not receive the latest delta CRL, it would assume that the certificate of the other entity is still valid. CRN is a valid certificate with revocation notice appended and resigned by the CA. This means that when a client wants to get the certificate of another entity, it will also get the revocation notice immediately. This causes a problem that a certificate cannot be stored within a domain CS (caching).
- Certification Path: An arbitrary number of CAs must validate each other until the certificate is obtained. Certification path validation is used when several CAs, which are not all connected to each other via a cross-certification, validate each other until the certificate is obtained.
- Cross Certification: Cross certification is used to create the certificate between two CAs. If both CAs trust each other then a cross certificate pair is established.
- Key Generation: There are at least two ways of generating the keys (Centralised and Basic Authenticated), but the common steps to be performed are:

(a) Subject identification.

(b) CA sends secret information (usually offline) to the subject.

(c) The key pair generation is performed (by the CA or by the subject).

(d) Connection is established between CA and Subject (using the secret information to ensure privacy) and either sends the public key to the CA or the private key is sent to the subject.

(e) CA sometimes sends the certificate to the user asking for an acknowledgment.

(f) The disadvantage of sending private keys is the loss of the no repudiation function.

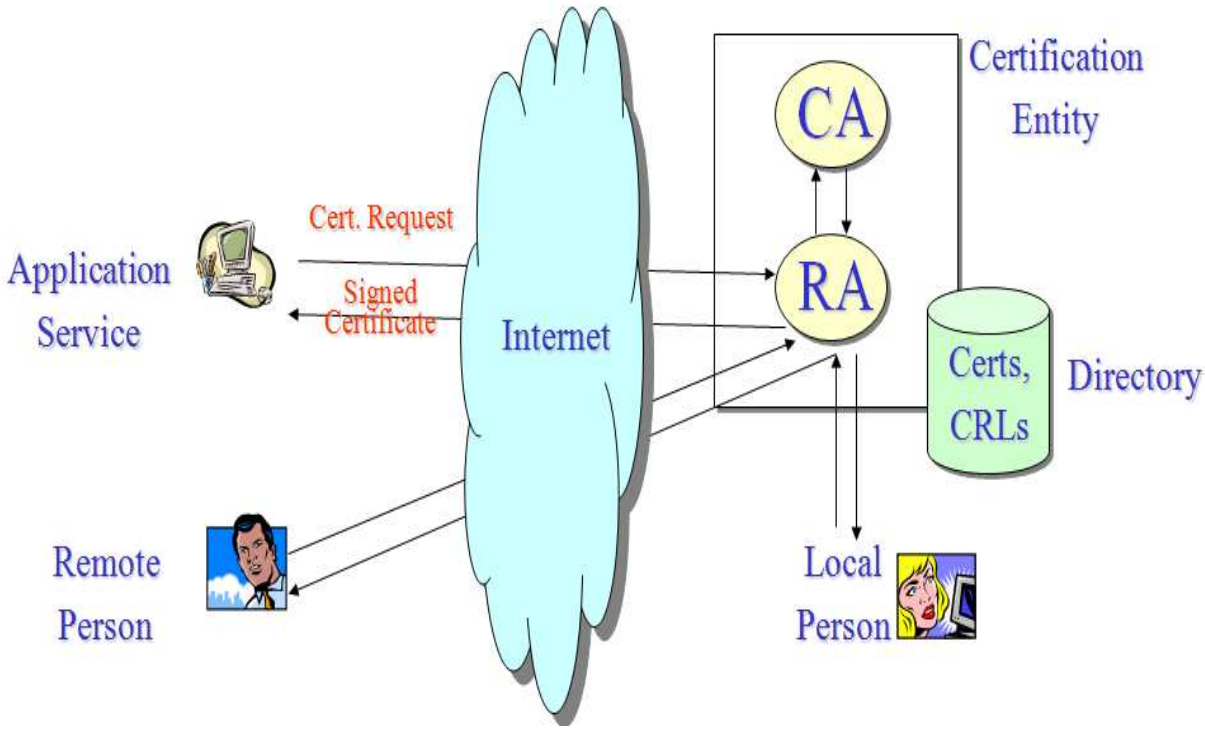


Diagram explaining how PKI works [This was obtained from ref [42]].

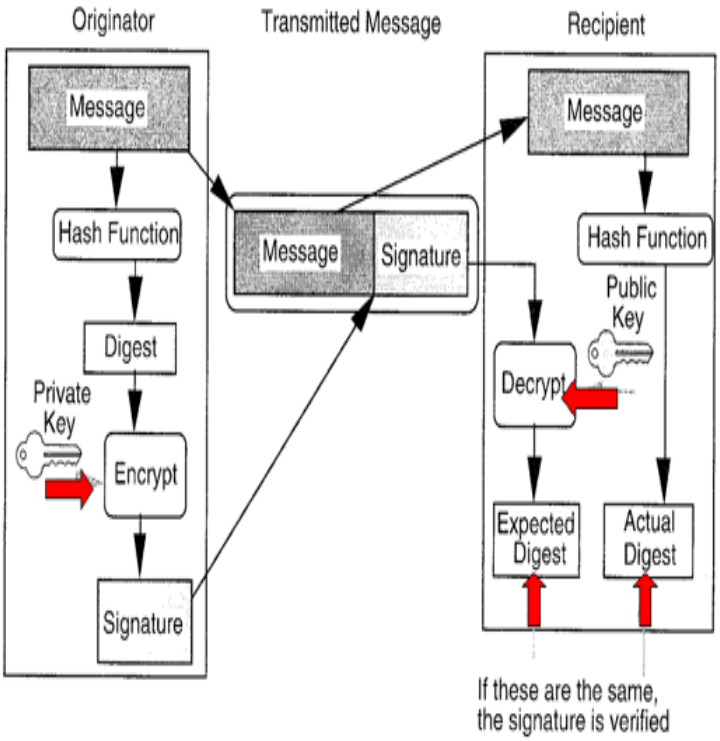
2.2 How PKI Works and Example

PKI is based on two systems, the Elliptic Curve Cryptosystem (ECC) and the modular exponentiation (RSA) system. The ECC system is usually referred to as a suitable solution for devices with limited resources because of its efficiency and the small amount of keys required to attain security [8]. However, conventional PKI such as PKIX and wireless PKI (WPKI) are based on modular exponentiation (RSA) which is not so suitable for systems with limited resources [11]. The ECC solution's security can be proved based on the difficulty of solving the Elliptic Curve Discrete Logarithm Problem (ECDLP) and factorisation in the elliptic curve [9]. The EC-based system can attain a substantial level of security with significantly small keys than its modular exponentiation

based counterpart e.g. it is believed that a 160-bit key in an elliptic curve based system provides the same level of security as that of a 1024-bit key in the RSA-based system [9]. As a result of this, it gives great efficiency in key storage, certificate size, memory usage and reduced processing. Therefore, it enhances the speed, leads to efficient use of power, and bandwidth in a resource constrained system.

PKI operation is based on certificates. The CA issues a user with a certificate containing a public key. The public key is used to encrypt data while the associated private key is used to decrypt. The private key is also provided to the user. This is kept secure so that no one would have access to it. The private key is also used to create a digital signature which could be confirmed by the public key.

Key Function	Key Type	Whose Key Used
Encrypt data for a recipient	Public Key	Receiver
Sign data	Private Key	Sender
Decrypt data received	Private Key	Receiver
Verify a signature	Public Key	Sender



How PKI Works in Digital Communication [This was obtained from ref 12]

PKI has been incorporated in many applications. For example, e-passport [13]. Using the digital signature, the receiving countries can verify that the stored data is authentic (i.e. generated by the issuing country and not altered). Public key cryptography is used to digitally sign the data stored on the chip. To Incorporate PKI with e-passport, the PKI must focus on the following:

- The adoption of the proper trust architecture
- The legal status of the certification provider
- Key and certificate management
- Interoperability and technology used.

Digital signature is used to verify the authenticity of data in e-passport. The mechanism of signing and validating data is called 'passive authentication' [13]. The signature algorithms used are RSA, DSA and Elliptic Curve DSA (ECDSA). The overhead cost of incorporating PKI in e-passport is relatively expensive because each country requires its own CA. ICAO (International Civil Aviation Organisation) requires two trust levels. The root CA is called Country Signing Certification Authority CSCA and the other called Document Signing Certification Authority, DSCA. There is no Single Point-Of-Trust (SPOT) and not even a European Union CA. It is not globally acceptable for political reasons.

Key management is also a big issue, the e-passport securely creates and hosts the private key, and the public key is a part of the signed data therefore it is bound to the identity details. The authentication procedure is based on a challenge-response mechanism where the passport proves the possession of the private key. This method is not fully implemented therefore it is weak and vulnerable to eaves-droppers.

Key revocation is almost unattainable because there is no means to revoke an AA key. In case of passport loss or compromise, the key cannot be changed during the lifespan of the passport. Obtaining a certificate in PKI is relatively expensive.

2.3 Password Authenticated Key Exchange

PAKE protocols enable two users to generate and exchange messages based on a common, cryptographically strong key which has an initial low-entropy secret (i.e. a password) which they share. The purpose of PAKE is to provide a strong mutual password authentication in which the only way an adversary would have access to it is by running an online dictionary attack. Online attack simply means an attacker iteratively guessing passwords and requires feedback from the server. The attacker does this by impersonating one of the users. Note that online attack is easy to detect and stopped since it interacts with a login server which is active. Using PAKE, the authenticating parties

can "bootstrap" a short secret (the password) into a long secret (a cryptographic key) that can be used to provide a secure channel [17]. Various protocols have been proposed which proves that PAKE is secure against offline dictionary attack. Offline dictionary attack is one in which the adversary does not require a feedback from the server. The adversary would have to eavesdrop honest execution and later try various passwords offline. Most protocols have the notion of Key Secrecy (which means that no computationally bounded adversary should learn anything about the session key shared between two honest users) and Forward Secrecy (which means that even if the long-term secret key of a user is being compromised, the adversary does not have access to the previous session keys).

2.3.1 Approaches to Password Authenticated Key Exchange

Unlike the password authenticated key exchange discussed below, most common techniques for password authentication are unilateral authentication techniques. That is, only one party (a user or client) is authenticated to the other party (a server) but not vice-versa. It requires the user sending its password to the server for verification. This technique is vulnerable to offline dictionary attack or relies on certified/authenticated public keys. It is currently used in web services as well as older applications. It is obviously insecure against eavesdroppers on the network because an attacker can impersonate the client but it's still considered acceptable on channels on which eavesdropping is relatively difficult.

A more advanced technique is challenge-response. A client sends the password to the server then the server sends a challenge message to the client to verify, based on the response it could determine if the client is valid or invalid. For instance the hash of the challenge and password involved. This type of authentication technique is used in some operating systems to enable network access. It is vulnerable to offline dictionary attack by an eavesdropper, since the challenge and its corresponding response, together makes password verification information.

Another secure technique sends a password to the server over an anonymous secure channel in which the server has been verified using a public key. This type of authentication is used in some remote terminal applications as well as web based applications [17] and it depends solely on the ability of the client to verify the server's public key else the server can be impersonated by an attacker. When used on the web, the public key of the server is certified by a Certification Authority that is trusted by the client. For remote terminal applications, there is no trusted third party and security relies on the client recognizing the public key, probably with the hash of the public key or something similar.

2.3.2 Evaluation of other PAKE protocol

Bellovin and Merrit introduced the first PAKE protocol in the year 1993 called A-EKE [14]. In A-EKE, H(P) is used for the storage of password and both parties use it as their secret. The symmetric encryption required *p* very close to 2^n where *n* is a bit-length of *p* and it required random data to be padded to fill the block size [16], a generator [19] and a digital signature scheme such as ElGamal [15] or p-NEW Signature [18] were required for this protocol and also for the security. A-EKE requires a lot of exponentiation operation and protocol steps, it also requires a safe prime modulus.

Jablon proposed B-SPEKE in 1997 [23, 24]. It is the verifier-based augmentation of SPEKE. It required relatively heavy (parallel) exponentiations and needs a server to choose another random number and a safe prime modulus. The protocol required six message exchange and four parallel exponentiation [24].

SRP (Secure Remote Password) protocol was proposed by Wu in 1998 [28]. It is a password authentication and key-exchange protocol suitable over an untrusted network. SRP was the most efficient password based protocol. It did not agree with the Diffie-Hellman exponential and was not in favour of generalisation especially in the elliptic curve group.

Kwon and Song proposed GXY protocol in 1999 [25]. It was derived from the SRP protocol but it agreed with the Diffie-Hellman exponentially g^{xy} as a keying material via password authentication [25]. GXY was not in favour of generalisation and it was less efficient than SRP. This protocol secures a weakly-chosen password and prevents password file compromises.

Mackenzie and Swaminathan introduced the first provable verifier-based protocol called SNAPI-X in 1999 [29]. It was the augmented version of SNAPI. Its proof of security is under the RSA and Diffie-Hellman scheme. In this protocol, the server stores a one-way function of the password and an adversary cannot impersonate a client. It has constraints in choosing specific parameters and also has export/patent restrictions.

Bellare and Rogaway introduced AuthA protocol in 2000 [20]. AuthA protocol has lower cost in communication and greater versatility. The method is based on the encrypted Diffie-Hellman key exchange. It has many choices possible for the underlying group. When it is an elliptic curve group, the communication cost for AuthA is as little as two flows of (approximately) 160 bits and one flow of (approximately) 64 bits [20]. The client and server each perform three exponentiations (multiplication in the language for elliptic curve groups), with trivial computational overhead beyond that. For both client and server, one of these operations may be done off-line.

Boyko, Mackenzie and Patel introduced PAK-X protocol in 2000 [26]. The protocol has its proof in the ROM (Random Oracle Model) and Diffie-Hellman assumption. In this protocol, the client stores a plaintext version of the password, while the server stores a verifier for the password. It is not efficient in its execution time because it does not provide parallelism to each party.

Kwon proposed a PAKE protocol called AMP in 2000 [21]. It is provable secure in the ROM. It has light constraints and it has several variants for accommodating implicit verifier.

The table and diagram below shows the performance level of each password protocols.

	Protocol	*Large*	*Exponentiations*			*Random Numbers*	
	Steps	*Blocks*	*Client*	*Server*	*Parallel*	*Client*	*Server*
A-EKE	7 (+4)	3 (+1)	4 (+2)	4 (+2)	6 (+3)	1 (+0)	1 (+0)
B-SPEKE	4 (+1)	3 (+1)	3 (+1)	4 (+2)	6 (+3)	1 (+0)	2 (+1)
SRP	4 (+1)	**2** (+0)	3 (+1)	3 (+1)	4 (+1)	1 (+0)	1 (+0)
GXY	4 (+1)	**2** (+0)	4 (+2)	3 (+1)	5 (+2)	1 (+0)	1 (+0)
SNAPI-X	5 (+2)	5 (+3)	5 (+3)	4 (+2)	7 (+4)	2 (+1)	3 (+2)
AuthA	**3** (+0)	**2** (+0)	4 (+2)	3 (+1)	6 (+3)	1 (+0)	1 (+0)
PAK-X	**3** (+0)	3 (+1)	4 (+2)	4 (+2)	8 (+5)	1 (+0)	2 (+1)
AMP	4 (+1)	**2** (+0)	**2** (+0)	**2** (+0)	**3** (+0)	1 (+0)	1 (+0)

Table 1: Comparisons of Verifier-based Protocols [This was obtained from ref 21]

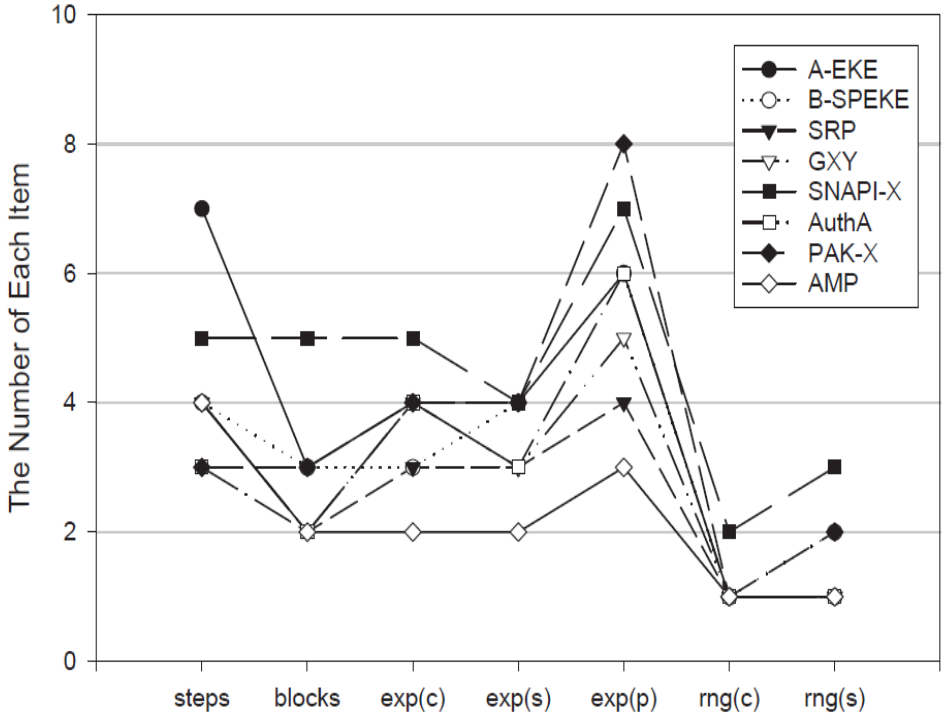


Figure 6. Graphical Representation of Table 1 [This was obtained from ref 21]

2.3.3 On-line and Off-line Attacks

An On-line attack is a serial exhaustive search for a secret, it is performed on-line using a server that verifies the secret. It is a scenario whereby the attacker require a feedback from the server when launching its attack. Considering a password-based challenge response protocol where a server gives a random challenge (r) to a client, and then the client returns the server res := $E_{pass}(r)$, the encryption of r using a pre-shared (hashed) password pass as its symmetric key. An adversary in the on-line attack runs a protocol with the server impersonating the client and then tries guessed passwords pass' on-line returning res' := $E_{pass'}$ (r) to the server. If it is accepted, then pass' is the target password with the highest probability. This attack can be prevented by letting the server take appropriate intervals between invalid trials e.g. timestamps, MAC.

An Off-line attack is that which is performed off-line in parallel using recorded transcripts of the protocol. It is one in which the attacker does not require a feedback from the server when launching its attack. Adversaries in the off-line attack can first obtain valid pairs of (r) and (res) by eavesdropping honest executions of the protocol, and then finds pass' satisfying res= E_{pass}' (r) off-line in parallel. Since the attack is performed off-line in parallel and the entropy of a password is usually not large enough, they can find it after series of exhaustive search.

While the on-line attack can be prevented by letting the server takes appropriate intervals between invalid trials, the off-line attacks cannot be prevented by such measures since the attack is performed off-line and independently of the server. Thus, the off-line attack is critical to most of the protocols using human-memorable passwords not having enough entropy to avoid off-line exhaustive search.

Off-line attack is also applicable to DH-EKE (Diffie-Hellman Encrypted Key Exchange). If the principal group size $log_2|G| = log_2 q$ is smaller than the encryption size (note, the condition is usually true when a prime order subgroup and conventional stream cipher or block cipher such as AES are used). DH-EKE is a protocol in which two entities $y_1 = E_{pass}(g^{r1})$ and $y2 = E_{pass}(g^{r2})$ respectively, and then share $D_{pass}(y_1)^{r2} = D_{pass}(y_2)^{r1} = g^{r1:r2}$ as a fresh secret where g is a generator of a finite cyclic group G=<g>; $E_{pass}()$ and $D_{pass}()$ are encryption and decryption functions using a hashed password pass as its symmetric key. The off-line attack on DH-EKE is as follows: Adversary A obtains some $y_1$and y_2 by eavesdropping the protocol and then goes off-line to see whether $D_{pass'}(y_1)$ and $D_{pass'}(y_2)$ are right members in G for guessed passwords pass'. If at least one of them is not a right member then the guessed password is wrong. By continuing the process above, the adversary could have access to the correct password.

2.4 The Protocol

This is a 3 round PAKE protocol with human-memorable password. It is a protocol which relies on a number or building blocks for its security. Firstly it uses the Cramer-Shoup cryptosystem [22] which is secure under adaptive chosen-ciphertext attack. Secondly it needs a one-time signature scheme [27] secure against existential forgery. Finally its proof of security is under the Decision Diffie-Hellman assumption [4]. It requires only (roughly) 8 times more computation than "standard" Diffie-Hellman key exchange [4] (which provides no authentication at all). It is assumed that the public parameter is known by all parties. The protocol works in the standard model only; it does not require a "random oracle" assumption. It also considers human input as the password is chosen from a relatively small set of passwords and it's secure too. The protocol relies on public key technique (this is necessary) but it is not a "public key solution" [31]. In particular, the participants do not require any public key, but instead rely on one set of common parameters shared by everyone in the system. This avoids problems associated with PKI (such as revocation, centralised trust, key management issue e.t.c). Furthermore, no participants know the secret key associated with the public parameters. This eliminates the risk that compromise of a participant will compromise the security of the entire system.

Proof of Security: A protocol that has its prove of security in the standard model (as in the protocol) is said to be secure because it has its cryptographic scheme based on complexity assumption

for example; [30] given (g^x , g^y, g^z) and (g^x, g^y, g^{xy}) at random from a group {1,.....|G|} it is impossible to distinguish between the two distribution.

In the Random Oracle Model (ROM) it is assumed that some hash function is replaced by a publicly accessible random function "the random oracle". This means that an adversary cannot compute the result of the hash function without querying the random oracle. The ideal cipher model is somehow similar to ROM but instead it has a publicly accessible random block cipher or ideal cipher.

Cramer Shoup cryptosystem [22] is an asymmetric key encryption algorithm and it is secure against adaptive chosen ciphertext attack. It uses a standard cryptographic assumption and its security is based on DDH assumption. It adds elements to ensure non-malleability. Non-malleability is achieved through "collision-resistant hash function". It consists of 3 algorithm key generation, encryption and decryption algorithms.

Key gen:

- Alice generates two distinct random generator $g_1, g_2 \in \boldsymbol{G}$ of order q
- She chooses random values $(x_1, x_2, y_1, y_2, z) \in \mathbf{Z}_q$
- Computes $c=g_1^{x1} g_2^{x2}$, $d=g_1^{y1}g_2^{y2}$, $h=g_1^{z}$
- Publishes (c, d, h) along with g_1, g_2 as public key and retains (x_1, x_2, y_1, y_2, z) as its secret key

Encryption:

To encrypt a message m $\in \boldsymbol{G}$ to Alice, Bob chooses a random k $\in \mathbf{Z}_q$ and computes the following:

- $u_1=g_1^{k}$, $u_2=g_2^{k}$
- $e=h^k m$
- $\alpha=H(u_1, u_2, e)$ where H () is a collision resistant cryptographic hash function.
- $v=c^k d^{k\alpha}$
- Bob sends ciphertext (u_1, u_2, e, v) to Alice.

Decryption:

To decrypt the ciphertext, Alice uses her secret key and computes the following:

- Computes $\alpha=H(u_1, u_2, e)$ and
- verifies that $(u_1^{x1} u_2^{x2})^{\alpha}(u_1^{y1}u_2^{y2})^{\alpha} = v$ if this test fails then the ciphertext is rejected; if otherwise, the plaintext is computed as $m = e/u_1^{z}$

The decryption works since $u_1^{z} = g_1^{kz} = h^k$ and $m = e/h^k$.

2.4.1 Protocol Initialization

The protocol requires a fixed set of participants (principals) each of which is either a client $C \in$ Client or a server $S \in$ Server (client and server are disjoint). Each C ∈ Client has a password pw_c and each S ∈ Server has a vector $PW_s = \langle pw_c \rangle c \in$ client, which contains the passwords of each of the client (the client shares password with the server). Recall that pw_c is what the client uses to log in; therefore it is assumed to be chosen from a relatively small space of possible password.

Before the protocol is run, an initialization process takes place during which public parameters are set and passwords are chosen for each client. The password for each client is chosen independently and uniformly at random from a set {1,…, N} where N is a constant, independent of the security parameter.

2.4.2 Protocol Execution

Let p, q be primes such that $q|p - 1$, and let $\mathcal{G}$ be a subgroup of $\mathbb{Z}_p^*$ of order q in which the DDH assumption holds. During the initialisation phase, generators $g_1, g_2, h, c, d \in \mathcal{G}$ and a function $\mathcal{H}$ from a family of universal one-way functions [32] (which is based on any one-way function) are chosen at random and published. Note that this public information is not an added assumption; "standard" Diffie-Hellman key exchange assumes that parties use a fixed generator g although this is not necessary, and [26,33] requires a public generator g for their proof of security. This protocol requires that the discrete logarithm of any of the generators with respect to any other is known by no one, therefore a trusted party who generates the public information, or a source of randomness which can be used to publicly derive the information, is required.

In the initialisation phase, passwords are chosen at random for each client. It is assumed that all passwords can be mapped to $\mathbb{Z}_q$. For a typical value of |q|, this will be a valid assumption for human-memorable passwords.

Execution of the protocol is as follows: When client C wants to connect to server S, the client first runs the key generation algorithm for the one-time signature scheme, giving VK and SK. Then, the client computes a client-encryption of $g_1^{pw_c}$. This, along with the client's name is sent to the server as the first message. The server chooses random elements x_2, y_2, z_2, w_2 from $\mathbb{Z}_q$ computes α' using the first message, and forms $g_1^{x_2}\, g_2^{y_2}\, h^{z_2}\, \left(cd^{\alpha'}\right)^{w2}$ .The server then computes a server-encryption of $g_1^{pw_c}$. This is sent back to the client as the second message. The client selects random elements x_1, y_1, z_1, w_1 from $\mathbb{Z}_q$ computes β ' using the second message, and forms $k = g_1^{x_1}\, g_2^{y_1}\, h^{z_1}\, \left(cd^{\beta'}\right)^{w2}$. Finally, β 'and K are signed using the signing key which was generated in the first step. The *sid* is defined as the transcript of the entire conversation.

Public information: $p, q, g_1, g_2, h, c, d, \mathcal{H}$

Client		Server
$(VK, SK) \leftarrow SigGen(1^k)$		
$r_1 \leftarrow \mathbb{Z}_q$		
$A = g_1^{r_1}; B = g_2^{r_1}$		
$C = h^{r_1} g_1^{pw_c}$		
$\alpha = \mathcal{H}(Client\|VK\|A\|B\|C)$		
$\mathcal{D} = (cd^{\alpha})^{r_1}$		
	$Client\|VK\|A\|B\|C\|D \longrightarrow$	
		$x_2, y_2, z_2, w_2, r_2 \leftarrow \mathbb{Z}_q$
		$\alpha' = \mathcal{H}(Client\|VK\|A\|B\|C)$
		$E = g_1^{x_2} g_2^{y_2} h^{z_2} (cd^{\alpha'})^{w2}$
		$F = g_1^{r_2}; G = g_2^{r_2}$
		$I = h^{r_2} g_1^{pw_c}$
		$\beta = \mathcal{H}(Server\|E\|F\|G\|I)$
		$J = (cd^{\beta})^{r_2}$
	$\longleftarrow Server\|E\|F\|G\|I\|J$	
$x_1, y_1, z_1, w_1 \leftarrow \mathbb{Z}_q$		
$\beta' = \mathcal{H}(Server\|E\|F\|G\|I)$		
$K = g_1^{x_1} g_2^{y_1} h^{z_1} (cd^{\beta'})^{w1}$		
$Sig = Sign_{sk}(\beta'\|K)$		
	$K\|Sig \longrightarrow$	
		if $Verify_{vk}((\beta\|K), Sig) = 1$
		$C' = C/g_1^{pw_c}$
$I' = I/g_1^{pw_c}$		$sk_s = K^{r_2} A^{x_2} B^{y_2} (C')^{z2} D^{w_2}$
$sk_c = E^{r_1} F^{x_1} G^{y_1} (I')^{z1} J^{w_1}$		$else\ sk_s \leftarrow \mathcal{G}$

Fig.1. The protocol for password-AKE.[3]

In the protocol, the client and server perform validity checks on the messages they receive to be sure of the validity. In particular, each sides checks that the values they receive are actually in the group $\mathcal{G}$ and not the identity (it is required to check that the group elements indeed have order q). Validity checks are crucial, even chosen-ciphertext security of the underlying Cramer-Shoup cryptosystem implements it.

CORRECTNESS. In the execution of the protocol, *C* and *S* calculate identical session keys. Note that in an honest conversation, $\alpha = \alpha'$ and $\beta = \beta'$. Then:

$$sk_c = (g_1^{x_2} g_2^{y_2} h^{z_2} (cd^{\alpha})^{w2})^{r1} g_1^{r_2 x_1} g_2^{r_2 y_1} h^{r_2 z_1} (cd^{\beta})^{r2w1}$$

And

$$sk_s = (g_1^{x_1} g_2^{y_1} h^{z_1} (cd^{\beta})^{w1})^{r2} g_1^{r_1 x_2} g_2^{r_1 y_2} h^{r_1 z_2} (cd^{\alpha})^{r1w2}$$

this proves that they are equal.

2.4.3 Security of the protocol

The following theorem indicates that the protocol is secure, since all lower order terms are negligible in k (Appendix A has the definition for lower order terms).

Theorem. *let P be the protocol of Figure 1, where passwords are chosen from a dictionary of size N, and let k= |q| be the security parameter. Let A be an adversary which runs in time t and asks* $q_{execute}$, q_{send}, *and* q_{reveal} *queries to the respective oracles. Then:*

$$Adv_{P,A}^{ake} < \frac{q_{send}}{2N} + 2q_{send}\varepsilon_{sig}(k,t) + 2\varepsilon_{ddh}(k,t) + 2q_{send}\varepsilon_{cs}(k,t,q_{send}/2) + 2q_{send}\varepsilon_{hash}(k,t) + \frac{\min\{2q_{reveal}, q_{send}\}}{q} + \frac{2\min\{q_{reveal}, q_{execute}\}}{q^2}$$

Note that the Execute oracle does not give any information to the adversary. This is because Diffie-Hellman key exchange is the basic proof of security for this protocol, and it's secure under a passive attack.

For an adversary to perform an “impersonation attack”, the adversary would have to send the first and third messages (to determine the eventual session key of the server); if impersonating the client. When impersonating the server (to determine the eventual session key of the client), the adversary must “prompt the client to generate the first message and must send the second message. This is because

the protocol has 3 flows. Consider an adversary impersonating a client and the first message it gives is $\langle Client|VK|A|B|C|D\rangle$. We say the message is valid if:

$$log_{g_1} A = log_{g_2} B = log_h\left(C/g_1^{pw_c}\right) = log_{cd^{\alpha'}} D \qquad (1)$$

Where $\alpha' = \mathcal{H}(Client, VK, A, B, C)$, and pw_c is the password for *Client*. Similar proof is used to check the validity of the second message from an adversary impersonating a client (here the password which determines validity depends upon the name of the client to which the adversary sends the message). Notion of validity for the third message is not defined. The following fact is central to the proof:

Fact. *When an invalid message is sent to an instance, the session key computed by that instance is information-theoretically independent of all messages sent and received by that instance. This holds for both clients and servers.*

Proof. Consider the case of an adversary interacting with a server, with the first message as above. Let $\theta_1 \stackrel{\text{def}}{=} log_{g_1} g_2$; $\theta_2 \stackrel{\text{def}}{=} log_{g_1} h$; and $\theta_3 \stackrel{\text{def}}{=} log_{g_1}\left(cd^{\alpha'}\right)$. Consider the random values x_2, y_2, z_2, w_2 used by the server instance during its execution.

Element E of the second message constrain these values as follows:

$$log_{g_1} E = x_2 + y_2\theta_1 + z_2\theta_2 + w_2\theta_3 \qquad (2)$$

The session key is calculated as K^{r_2} multiplied by $sk_s' = A^{x_2} B^{y_2} (C/g_1^{pw_c})^{z_2} D^{w_2}$.

The protocol has:

$$log_{g_1} sk_s' = x_2 log_{g_1} A + y_2\theta_1 log_{g_2} B + z_2\theta_2 log_h\left(C/g_1^{pw_c}\right) + w_2\theta_3 log_{cd^{\alpha'}} D \qquad (3)$$

When equation (1) does not hold (i.e. the message is invalid), equation (2) and (3) are linearly independent and $sk_s' \in_R \mathcal{G}$ is information-theoretically independent of the transcript of the execution. A similar argument holds for the case of an adversary interacting with a client.

Let Π_U^i be an instance to which the adversary has sent an invalid message. The fact stated above implies that the adversary has advantage 0 (no advantage) in differentiating the session key generated by this instance from a random session key. Thus, for an adversary to have a non-zero advantage, it would have to send a valid message to an instance.

2.5 Evaluation of PKI against PAKE

PKI consists of a lot of components, this includes the CA, RA, digital certificates, certificate revocation e.t.c that are involved in the issuing, storing and/or distributing of certificates. It is also a distributed database of public key certificates and other informations attached to it.
The security of a CA is based on a lot of things. The cryptographic assumption is not all that is needed, it also requires the protection of the system that serves as the CA and the people involved. If a system that serves as the CA is infected by computer virus then a lot of havoc is caused.
Computer virus is a program that interferes with the normal operation of a system; it is capable of deleting and corrupting data on a computer system. If the CA system is corrupted then the certificates produced could be affected.

The verifying computer in PKI is also a thing to consider. The keys (public and private) required in PKI need to be kept safe. It cannot be said that the keys would be hundred percent safe on a system, because anybody could login into the computer and get information from it. This can be prevented if the system (CA) is kept isolated and under surveillance. Certificate verification does not use a secret key, it uses a public key, and therefore there is no secret to protect. However, it uses one or more "root" public keys therefore if an attacker add his own public key to the list, then he can issue his own certificates, which will be treated exactly like the legitimate certificate. The best practise to prevent this attack is to do all certificate verification on a computer system that is invulnerable to penetration by hostile code or to physical tampering [34].

PKI is correct if it allows an entity to conclude about another entity correctly, and disallows it if its conclusion is wrong. Therefore PKI designers need sophisticated tools which can accurately evaluate what type of trust judgement their system enables. Certificates contain more than names; they contain extensions, use policies, attributes e.t.c. Some types of certificates (e.g. X.509 attribute certificate [37]) bind a key to a set of properties and other types (e.g. SDSI/SPKI [35]) do not require names at all. In many real world PKI applications, a globally unique name is not even the relevant parameter [36]. Certificates are expected to expire and/or be revoked. Some systems use short certificate lifespan as a security advantage. Systems which use multiple certificates to describe an entity can have lifespan mismatch. For example, an attribute certificate that contains the courses entity A enrolled for in this term, may expire before her identity certificate. Some systems use a combination of multiple certificate types. The Grid community's MyProxy system uses X.509 certificates in conjunction with a short-lived Proxy certificates for authentication and dynamic delegation.

While PKI can realise an authenticated key exchange or key transport (which is secure against off-line attack) like SSH (secure shell), SSL/TLS (secure socket layer/transport layer security), station-to

station, we have to note that the receiver of a public key must verify them using the fingerprints (digest) of them or the verification keys of digital signatures attached with them [38]. This means the entity must carry about something which is hard to remember. On the other hand, PAKE protocols do not require it's entity to carry something hard to remember (apart from a password) to verify something.

In PKI, the main component involved is the Certification Authority. It is often referred to as the "trusted party". In cryptographic literatures, it only means it can handle its own private keys well, it doesn't mean it can be trusted when it comes to the validity of a certificate for a particular purpose e.g. making a payment or signing a purchase order. Having a CA is an additional cost in the implementation of security requirements. Before a user and server interact, they would have to first consult the CA to verify if the information is correct. In a case whereby the CA has been compromised, then the users are left to believe what they are being told even if it's false. But in the case of password, a trusted party is not needed. A user can communicate to the server via the password which they share.

There are problems involved in using PKI, this include cost, scalability (both of registration and key management processes), revocation, management of client private keys, and support for dynamic security policies. To operate and manage a PKI, there are a number of control areas which must be addressed by the CA:

- CA environmental control: These are processes, policies, procedures and technical control that create the secure and trustworthy environment for the CA. These includes
    (a) CPS (Certification Practise System) and CP (Certificate Policy) management.
    (b) Security management
    (c) Asset classification and management
    (d) Operation management
    (e) System access management
    (f) System development and maintenance
    (g) Monitoring and compliance

- CA key life cycle management: These are processes, procedures and technical control to maintain the security and integrity of CA keys throughout its life cycle. These includes
    (a) CA key generation
    (b) Key storage, backup and recovery
    (c) Public key distribution
    (d) Key archival and destruction
    (e) CA cryptographic hardware life cycle management

- Certificate life cycle management: These are processes, procedures and technical control concerning the management of certificates throughout its life cycle. These include
    (a) Subscriber registration
    (b) Certificate issuance
    (c) Certificate distribution
    (d) Certificate revocation and suspension
    (e) Certificate status information processing

In security perspective, more components generally imply more point of vulnerabilities. With all these to consider in the construction of PKI, these make the overhead cost of embedding PKI in applications expensive. While for password based, a user only need to memorize a short password and can be authenticated at anytime, anywhere, regardless of the type of access device used. At large, password has been the most pervasive user authentication means for a long time and is still gaining popularity even in the presence of several alternative strong authentication approaches like digital signature and biometrics. This is because password authentication requires no dedicated device, which is of great importance as users are increasingly roaming nowadays.

Related Works

The first formal treatments [31,39] were in a model in which participants already share some cryptographically-strong information: either a secret key which can be used for encryption of data/signing of signature and authenticity of message, or a public key which can be used for decryption of data/verifying of signature. In addition, other PAKE protocol [33, 26, 40] was implemented; [33] relies on ideal ciphers and [26, 40] rely on random oracle for their proof of security. None of them has their security proven in the standard model which is more attributed to the cryptographic assumption. Recently, Goldreich and Lindell [1] have shown a protocol based on general assumption, secure in the standard model with human memorable password. Their protocol requires no public parameters, it has a non-constant number of rounds and it requires techniques from general multi-party computation. Multi-party protocol is one that requires a “trusted party”. This makes the protocol good in theory but not in practise.

Chapter3: Implementation

3.1 Introduction

The Aim of this project is to implement a secure channel in a prototype Web Service. The implementation is centred on the protocol proposed in [3]. This is a password based protocol therefore it is required for the participants to have a common password only and does not have the issues attributed to PKI which currently exists. The protocol achieves Key Exchange and not mutual authentication [3]. This means that the session key at the client side must be equal to the session key on the server side. The protocol was tested using two prototype applications (InvestmentAdvisor and StockExchangeLatestPrice) which were developed in java and the exchange of messages was enabled through the use of a prototype Web Service developed. The project is written in java which is a popular programming language as against the proprietary languages which are popularly used on the internet.

The problem specification is stated in the next section followed by the implementation details, the challenges which were encountered, and what was done to resolve it.

3.2 Problem Specification

A Web Service is a technology that enables the communication and exchange of messages over the internet. Essentially, web services are technologies that enable communication and exchange of messages between applications of diverse platform over the internet. They aid companies and organisations in making real-time consumption/utilization of resources and software components outside their implementation domain. This way, they save capital and still achieve expected results with all required resources and functionality. In non public web services, requests are restricted to authorised companies or individuals. Confidentiality, Integrity and Authenticity of data are therefore paramount issues. To make use of them one needs a secure channel (which enforces the above requirements). Currently on the Web, secure channels are implemented using Public Key Infrastructure (PKI). An adversary could delete and insert messages because he may control the communication. In PKI, a third party “trusted” is needed which implies another security requirement. Also the user or server must interact with the third party before they can communicate remotely which increases the overhead of the system. In the concept of security, having an extra party is relatively expensive and should be avoided. PKI requires centralised trust, has key issues (key revocation, key management e.t.c) and deleting of the expired key from the database.

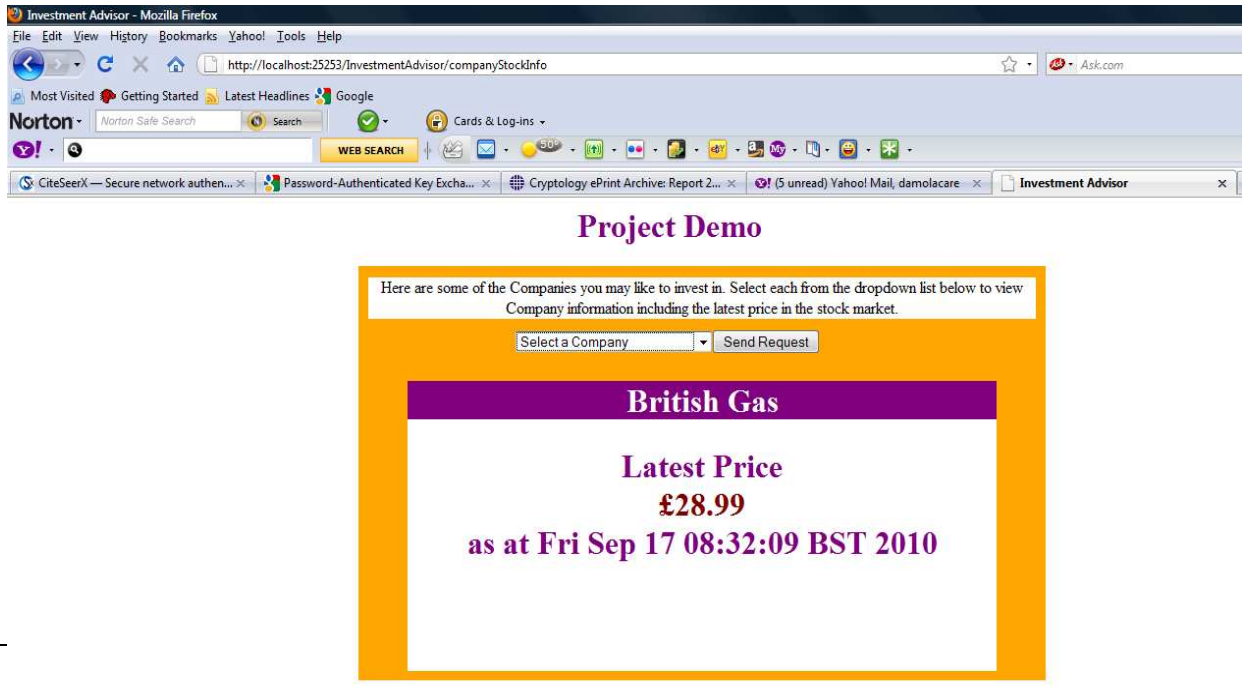
Despite all this there are other issues like the security of the trusted third party. This project involves the participants to know only their password thereby preventing the problems attributed to PKI.

3.3 Project Design

The design of the project is divided into stages. Firstly, the two applications (InvestmentAdvisor and StockExchangeLatestPrice) which were used for testing. Secondly, the prototype Web Service which aids the exchange and finally the protocol. There are various classes used in this project and each class is interconnected to achieve the overall outcome of the project.

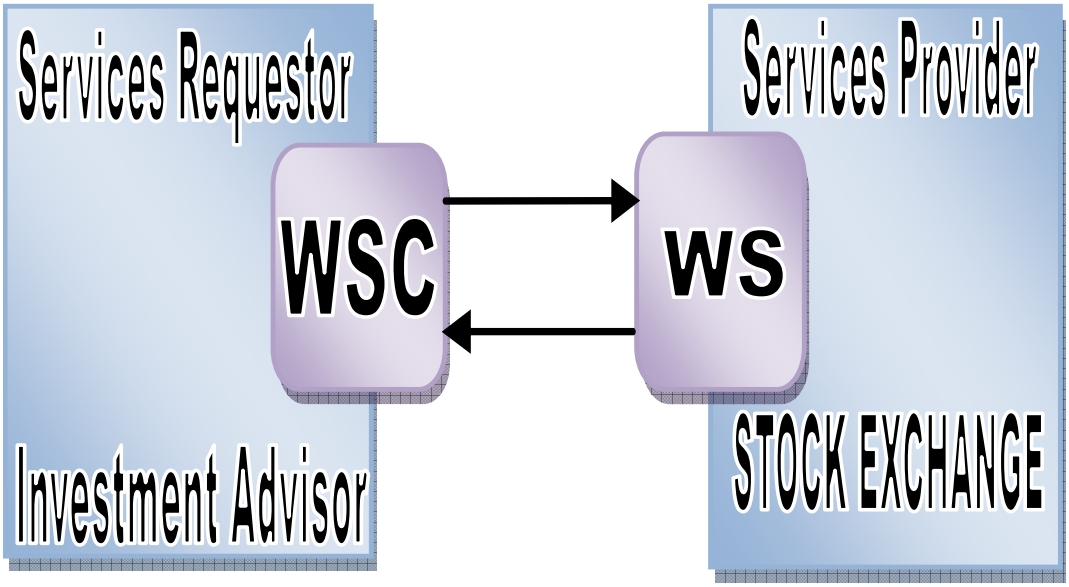
InvestmentAdvisor:

This is the user interface and a prototype application that was built for testing. It has lists of companies in the drop down list boxes. When any company is selected, it produces the stock price of the particular company chosen by the requester. This is a prototype non public application, because it is restricted for the companies in the drop down boxes only, examples are British Gas, BT, First Travel, HSBC, London Bar, Metaswitch, RBS and Virgin. When a company is chosen, it invokes the backend operations which involve the protocol exchanging messages to produce an equal session key. The key is then used to encrypt the value before sending it to the client. The client application then uses its own key to decrypt the information before displaying it on the screen for the requester. All the backend operations performed would be discussed in subsequent pages. If the password is not correct, it returns a null value.



StockExchangeLatestPrice:

This is the Web Server that hosts the application. It has a database that contains the stock prices of the company and constructor that invokes the protocol's class. It runs on Apache Tomcat 6.0.20. When a company is selected by a requester on the client, it invokes the companyStockInfo.java class which has a Servlet. A servlet is a java programming language class that when being invoked extends the functionality that resides in the server. It pulls-out the host application that resides in the server via the request-response programming model. Though it responds to any type of request, it is commonly used to extend the applications hosted by web servers. The **javax.servlet.http.HttpServlet**, **javax.servlet.http.HttpServletResponse** and **javax.servlet.http.HttpServletResponse** packages provide methods such as the get and post for handling HTTP-Services.



3.3.1 Computation of the protocol

The protocol is divided into two stages, the initialisation phase and the execution phase. The initialisation phase is where the public information, elements and password used during the execution phase is generated, while the execution phase involves all the computation and exchanges that takes place to generate equal session keys.

Initialisation:

Two prime values p & q are chosen such that q|p-1 is satisfied. If q= 5, p is generated by computing p=2*q+1 therefore p=11. After having the two prime's p & q, a group Z_p^* is generated.

$$Z_p^* = 2q$$

Therefore, $Z_p^* = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ in this example.

- To generate the group $\mathcal{G}$ which is a subgroup of Z_p^* , an element is chosen at random and squared. For example if 2 is chosen then we have $2^2 = 4$

Then, the elements of $\mathcal{G}$ are:
$4^0 \bmod 11 = 1$
$4^1 \bmod 11 = 4$
$4^2 \bmod 11 = 5$
$4^3 \bmod 11 = 9$
$4^4 \bmod 11 = 3$
$4^5 \bmod 11 = 1$

So the group $\mathcal{G} = \{1, 3, 4, 5, 9\}$ is a subgroup of Z_p^* . The values in the group are dependent on q. This means it iterates q times before it repeats itself. The values for the public information (p, q, g_1, g_2, h, c, d) in the protocol are derived from the group $\mathcal{G}$ at random.

- The elements (r_1, x_2, y_2, z_2, w_2, r_2 , x_1, y_1, z_1, w_1) from the protocol are in group $\mathbb{Z}_q$. This group is derived by

$$\mathbb{Z}_q = \{0, \ldots\ldots q\text{-}1\}$$

Therefore it is derived from whatever the value of q is minus 1. The elements are picked at random in this group.

- The password pw_c is mapped to $\mathbb{Z}_q$.

Execution:

After the client has been provided with the public information, elements and password, then they can perform the computations in the protocol. In order for the values to be generated, we have to perform multiplication in the group operation. Fig.1 omits a lot of implementation details which would be stated explicitly as possible in this section. From the protocol in fig. 1.

$$A = g_1^{r_1}$$

This is computed as

$$A = g_1^{r_1} \ mod \ p$$

The operation is called multiplication in the group. The result obtained from the exponentiation operation $g_1^{r_1}$ is being divided by the value of prime p. The remainder after this computation is the value for A. For example, if $g_1^{r_1} = 15$, remember that p=11.

Then the computation becomes

$$A = 15 \ mod \ 11$$

So the value of $A = 4$.

This format is used to compute the values of *A, B, D, F, G, J* in the protocol.

For efficiency, computations that have more than one exponentiation operation, the multiplication in the group operation is done intermittently. For example, from fig.1.

$$C = h^{r_1} \ g_1^{pw_c}$$

This is computed as

$$C = ((h^{r_1} \ mod \ p * g_1^{pw_c} \ mod \ p) \ mod \ p)$$

This could be computed anyway, the most important thing is for the overall computation to be in mod *p* but for efficiency purpose it was computed as above. This format is used to compute *C, E, I, K,* sk_c and sk_s

To compute the division in the group operation, we have to compute the inverse of $g_1^{pw_c}$ in the group (which is some other element of the group X such that X times $g_1^{pw_c} mod \ p$ equals 1) and then we multiply X with *C mod p*. For example,

C' = C/$g_1^{pw_c}$

Firstly, we have to find X such that X* $g_1^{pw_c} mod \ p$ =1. Assuming that $g_1^{pw_c}$ = 15 then we need to find a value such that

X*15 = 1 *mod p*

Therefore, 3*15 =1 mod 11

X=3

Then X * *C mod p*=3*629 mod 11=1887 mod 11= 6
therefore *C*'= 6.
This format is used to compute *C*' and *I*' called division in the group operation.

In the protocol, $\alpha = \alpha'$ and $\beta = \beta'$. This is obtained by concatenating the results from the computation and the names, then passing it to a Hash function (SHA-1). A hash function is a function that accepts an arbitrary number of inputs and produces a fixed amount of output. It produces a 160-bit output for every message that is passed across.
All the computation details and example are given to provide more information to the reader when reading the implementation part of this work. Having explicitly stated how each value in the protocol are generated, the next stage talks about the implementation

3.3.2 Implementation Details

Due to the extra cost of a third party in PKI, security of the third party, and the key issues related to it, another secure channel was looked at which is less expensive, proven secure and does not have key related issues. In order to verify if this protocol was realisable, two prototype applications involving stock price were built.
The investment advisor runs on a server called Glassfish v3. This was done because the public parameters and elements generated on the server were transferred to the client and had to be stored for subsequent executions. While the stock exchange latest price runs on Apache Tomcat 6.0.20 server. It contains the stock values.
The following section talks about the design consideration, Applications implemented and why.

3.3.2.1 Design Consideration

Based on the specification of the laptop used to implement the project
Processor: AMD Turion Dual-Core RM-72 2.10GHz
Memory (RAM): 4.00GB
System type: 32-bit Operating System
Operating systems: Windows Vista™ Home Premium
Service Pack 1
It was taking a lot of time to compute 512-bit primes. Due to this reason, 32-bit prime was used. If deployed into companies with a large server, processor and system type, then that shouldn't be a problem.

3.3.2.2 Applications

Some applications are similar to both the client and the server. This is because they perform the same function except that those in the server perform the function for the server, and those in the client perform the function for the client. These models include Public information provider, DB manager, Session key Encrypt Decrypt, Hashing, and Communication Manager.

Applications implemented on the Server

Protocol Initialisation:

This is responsible for the generation of the public information, elements and password that is used during the execution phase of the protocol. It follows the principle stated in (3.3.1) above to compute the parameters. It uses the BigInteger class embedded in JAVA. This is because BigInteger has unbounded properties which makes it contain very large values. The MillerRabin32 [41] function was also used to test the primality of prime values chosen. This is to be sure that the values of p & q chosen are actually prime numbers.

After the public parameters are generated, it publishes the information and then passes it to the DB manager. The DB manager then saves it to the DB (database). This was done because the values generated are needed in subsequent executions.

MillerRabin32:

This is an open source software obtained from the internet. It was gotten at "http://en.literateprograms.org/Special:Downloadcode/Miller-Rabin_primality_test_%28Java%29" solely because of its purpose. It is a program used to test the primality of the prime values chosen in our program.

ProtocolStep2:

The protocol requires different numbers of steps. Step 1 involves all the first computation done on the client's side before sending it to the server as the first message; step 2 involves all second computation done at the server's side before sending it to the client as the second message and so on. This protocolStep2 involves all the second computation in (fig.1.) following the principles in (3.3.1) before sending it to the client as the second message. It invokes the constructor of the public information provider class and the communication manager class in order to get the resources it needs during the computation.

After the computation is complete, the protocol message exchange application aids the sending of message to the client. This is be discussed later.

ProtocolStep4:

All the computation in the fourth stage of the server is performed here. The session key for the server is generated. It follows the same principle as above.

ProtocolMessageExchange:

It aids the exchange of the message with the client. This is achieved through the web service operation method and the get method which is embedded in web service. It invokes the constructor of protocolstep2 and protocolstep4. It gets the first message and stores it. As the first message it being received the second message is being sent automatically. The following algorithm shows how it is done.

```
@WebService()
public class protocolMsgExchange {
    @WebMethod(operationName = "getSecondMsg")
   public java.util.List<java.lang.String> getSecondMsg(@WebParam(name = "clientN")
        String clientN, @WebParam(name = "veriKey")
        int veriKey, @WebParam(name = "A")
        BigInteger A, @WebParam(name = "B")
        BigInteger B, @WebParam(name = "C")
        BigInteger C, @WebParam(name = "D")
        BigInteger D) {
        //store received first message in global store
    CommunicationManager.getInstance().storeFirstMsg(clientN, veriKey, A, B, C, D);
 //invoke ProtocolStep2 to compute the second message to be sent back to client
        ProtocolStep2 step2 = new ProtocolStep2("server");
       java.util.List<String> secondMsg = step2.getSecondMessage();
  //return second message to client
        return secondMsg;
    }

    /**
     * Web service operation
     */
```

```
    @WebMethod(operationName = "Verification")
    public void Verification(@WebParam(name = "k")
    BigInteger k) {
 //store third message in the global store
        CommunicationManager.getInstance().storeK(k);
//invoke ProtocolStep4 to finally use the third message to compute the session key
        ProtocolStep4 step4 = new ProtocolStep4();
   }
}
```

CommunicationManager:

When the first message is sent to the server, it also needs to store this information too because it would be invoked in subsequent executions. This was the motivation behind the communication manager. The communication manager is in charge of storing the previous executions. It stores the generated and exchanged information so that it can be used in subsequent executions when invoked. In order not to lose the previous ones, when the information is generated, it creates an instance of itself and makes it private so that it cannot be modified or changed by any other application. This application is available at the client's side and server side. The client's side communication manager stores the message that was sent by the server while the server side communication manager stores the message that was sent by the client.

There are some applications that are similar in the client and server. This is because they perform the same function in the client and server. The one that resides in the client performs the function for the client while the one that resides in the server performs the same function for the server. These include PublicInformationProvider, DB manager, Session Key Encrypt Decrypt, and Hashing.

Application Implemented on the Client:

PublicInfoProvider:

After the protocol initialisation phase is computed on the server, it saves the parameters into the DB (database) through the DB manager. For the client to have access to these parameters, the public information provider application invokes the getPublicInfoWS constructor to get the information

generated on the server and the loadPublicInfo constructor loads it to the DB manager of the client who eventually loads the parameters into the DB of the client. By doing this, the client also get to have a copy of the parameters needed for execution at its end.

ProtocolStep1:

The computations involving the first message are computed here. This is possible since the client now have access to the public information needed for the computation. It invokes the constructor of the public information provider class in order to get the resources it needs for the computation. Then it performs the computation that applies to it as stated earlier in (3.3.1).

After the computation is complete, the key exchange communication application aids the sending of message to the server. This would be discussed later.

ProtocolStep3:

It performs the computation involving the third and fourth message on the client.

KeyExchangeCommunication:

It aids the exchange of the message with the server. This is achieved through the web service operation method and the get method which is embedded in web service. It invokes the constructor of protocolstep1 and protocolstep3. It sends the first message. As the first message it being sent, the second message is being received automatically. The following algorithm shows how it is done.

```
public class keyExchangeCommunication {

        public keyExchangeCommunication(){

           ProtocolStep1 protocol = new ProtocolStep1 ("Damola");

         //send first message and receive second message in return

        List<String> secondMsg = sendFirstMsg (protocol.getClientName (),
protocol.getVeriKey (), protocol.getA (), protocol.getB (), protocol.getC (),
protocol.getD ());

        CommunicationManager.getInstance ().storeSecondMsg (secondMsg);

 new ProtocolStep3 ();

          thirdMsg (CommunicationManager.getInstance().getK());  //send third
message

     }

     public List<String> sendFirstMsg(String clientName, int VK, BigInteger A,
BigInteger B, BigInteger C, BigInteger D){

try {  // Call Web Service Operation

          Communication.ProtocolMsgExchangeService service = new
Communication.ProtocolMsgExchangeService();
```

```
            Communication.ProtocolMsgExchange port =
service.getProtocolMsgExchangePort ();

            //initialize WS operation arguments with method parameters

            java.lang.String clientN = clientName;

            int veriKey = VK;

            java.math.BigInteger a = A;

            java.math.BigInteger b = B;

            java.math.BigInteger c = C;

            java.math.BigInteger d = D;

                    java.util.List<java.lang.String> result = port.getSecondMsg
(clientN, veriKey, a, b, c, d);

            return result;

        } catch (Exception ex) {

            //handle custom exceptions here

        }

 return null;

 }

 /**

     * This method sends the third message to the server

     * @param K

     */

    public void thirdMsg(BigInteger K){

try { // Call Web Service Operation

            Communication.ProtocolMsgExchangeService service = new
Communication.ProtocolMsgExchangeService ();

            Communication.ProtocolMsgExchange port =
service.getProtocolMsgExchangePort ();

            // initialize WS operation arguments with method parameter

            java.math.BigInteger k = K;

            port.verification (k);

        } catch (Exception ex) {

            //handle custom exceptions here

}

     }

}
```

Hashing:

For alpha (α) to be computed, it needs to be passed to a hash function. A hash function (SHA-1) is a function that accepts an arbitrary number of inputs and produces a fixed amount of output. It produces a 160-bit output for every message that is passed across. The result produced is in an alphanumeric form and this cannot be used for other computation e.g. computation of D

In order to resolve this, the alphanumeric characters produced were converted to hexadecimal then to integer to enable its usage in subsequent computation. Without this conversion, it wouldn't have been possible for the values produced from the hash function to be used anywhere else.

For the data (stock price) to be passed securely over the internet, we decided to use the session keys generated to encrypt and decrypt the data. The session key for the server encrypts the data and the session key on the client can be used to decrypt it before it's sent on the screen.

AES-CBC mode of operation was used for this purpose. It performs encryption/decryption using 128-bit block size and key size ranging from 128, 192, and 256 respectively. Due to the fact that the session keys produced are not up to 128 bit, we decided to first pass the generated keys to MD5 which produces a 128-bit hash value. With this, it makes it possible for the session key to be used in the AES-CBC encryption/decryption operation. This was an additional input we realised during the research.

The Cipher Block Chaining (CBC) [43] mode is a confidentiality mode whose encryption process involves combining/chaining of the plaintext blocks with the previous ciphertext blocks. The CBC mode requires an IV to combine with the first plaintext block. In CBC encryption, each successive plaintext block is XORed with the previous output/ciphertext block to produce the new input block. The forward cipher function is applied to each input block to produce the ciphertext block.

In CBC decryption, to recover any plaintext block (except the first), the inverse cipher function is applied to the corresponding ciphertext block, and the resulting block is XORed with the previous ciphertext block. With this it is secure and an adversary would not have access to the data.

DB manager:

It has direct access to the DB (database). Any computation done is first sent to the DB manager then the DB manager sends it to the DB. No other application has direct access to the DB; they all need to pass through the DB manager. This was done for security purpose.

SessionKeyEncryptDecrypt:

The session keys generated are used to encrypt/decrypt values of the price being passed across the internet. This is done in order to prevent an adversary having access to the price when it's sent across. We used AES-CBC for this purpose. AES is a symmetric key encryption standard. The cipher produced is a 128-bit block size. When a company is selected, it initiates the whole protocol message exchange process and a session key is produced (which is equal for the server and the client). The session key is converted to a 128-bit by the MD5 function so it can be used by AES-CBC to perform the encryption/decryption function. The session key on the server is then used to encrypt the data before its being sent across, while the session key for the client is used to decrypt the message before it's placed on the screen. This is where the encryption/decryption is done.

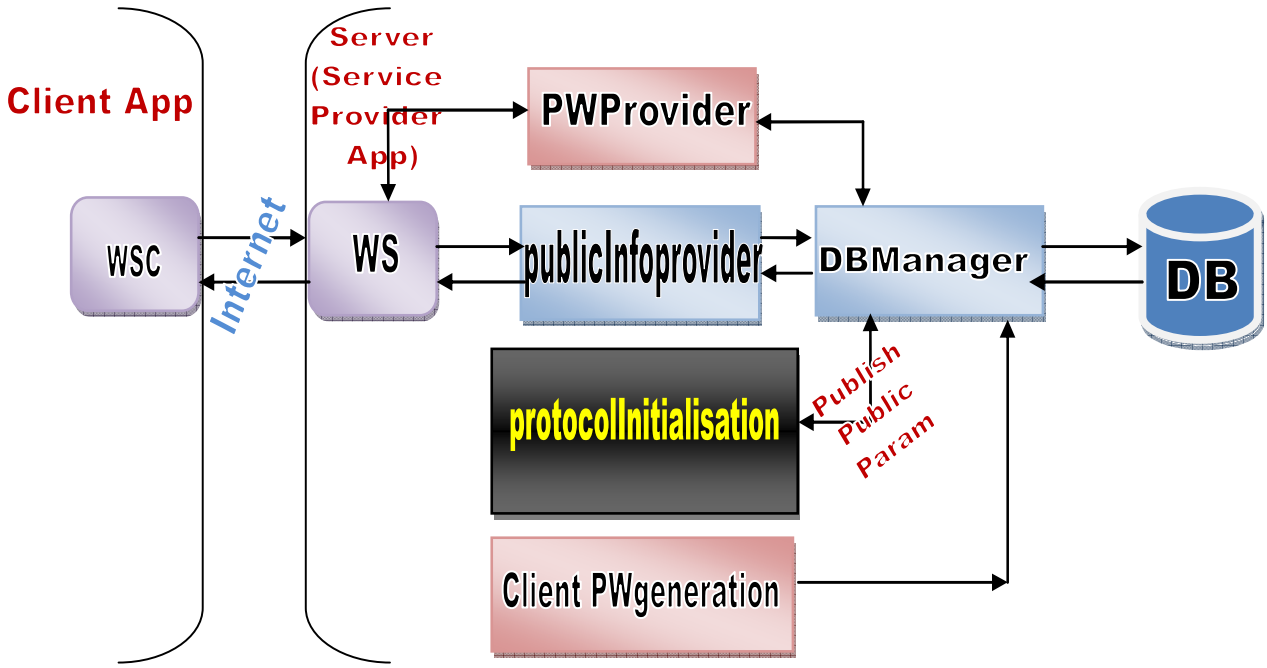
CompanyStockInfo:

This is the very first class that is invoked when a company is selected on the client's screen. It has a servlet as discussed earlier it pulls out the application in the host (web server). It processes the request-response method. So when a request is made, all the other processes are invoked and the result is eventually passed to it.

This section gives a detailed explanation of the work carried out. This is what motivated the design and the development of the software used to achieve the overall outcome of the project.

3.3.3 Diagram showing how the messages are passed

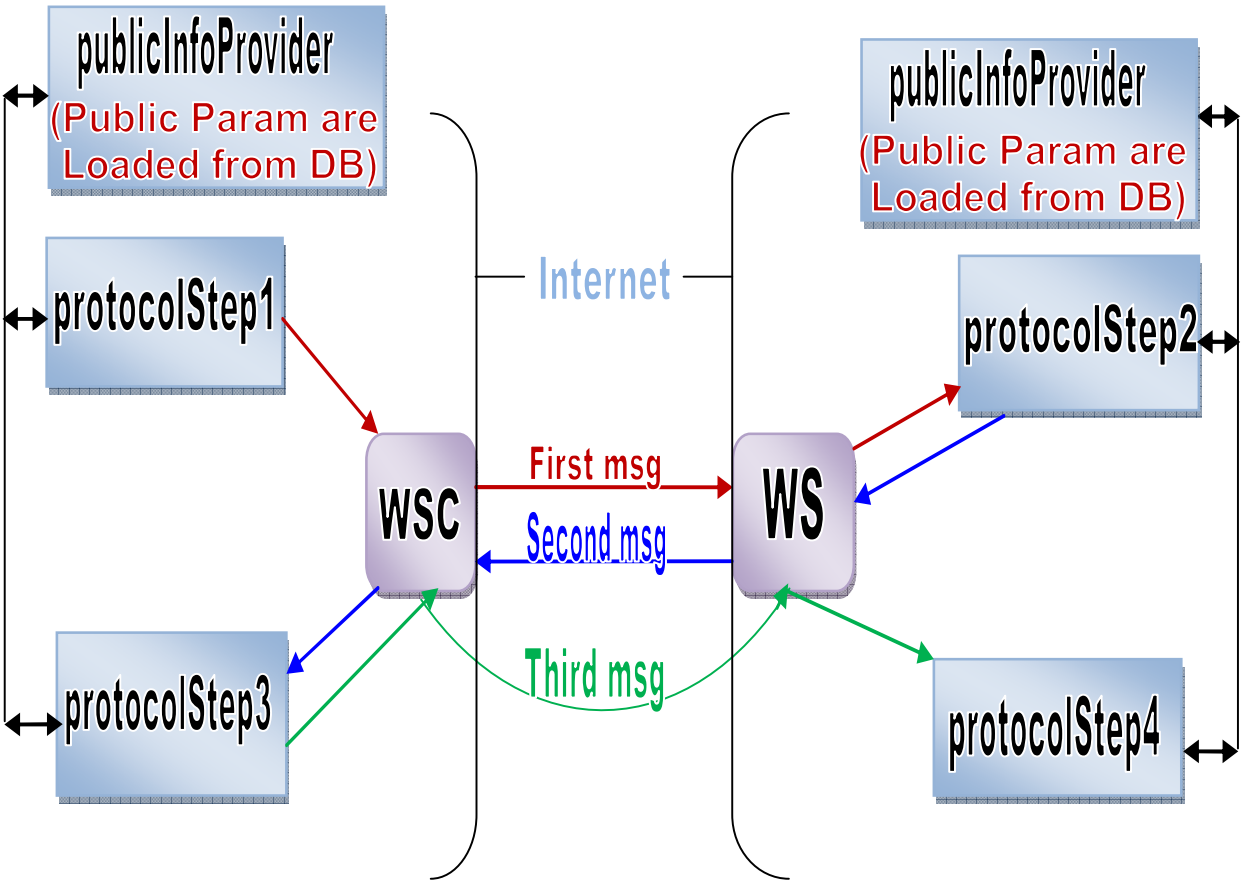
Initialisation phase:



The protocolInitialisation as stated above computes the parameters and transfers it to the DB manager to store it into the DB. The Client PWgeneration also performs its own function and transfers it for storage.

For the client to have the information generated on the server, the publicInfoProvider invokes the DB manager who gets the information from the database. It then transfers it back to the publicInfoProvider who later transfers it to the client through the Web Service operation. The PWProvider also follows the same process to get the password across to the client. With this, the client and server have access to the same information and eligible to do its own computation at its side.

Execution phase:



Now the client and server have access to the parameters needed in the execution phase. ProtocolStep1 follows the step as stated above to compute the first message that is being sent to the server through the web service. The second message is computed in protocolStep2 following the same principle

stated above and sends it as the second message to the client. ProtocolStep3 performs its own part and sends it as the third message. This is where the session key for the client is computed. Finally, ProtocolStep4 performs it own computation, the session key for the server is performed here. The ability of the message to be passed across is achieved through the web service on the client and server.

3.4 Implementation Challenges and Additional input

There were quite a number of challenges encountered during the implementation of the project. A lot of application had to be put in place for the efficiency of the protocol and making sure it works. Selected ones which required a lot of research are discussed below.

The protocol required prime values and large values at that. That kept us thinking on how to go about implementing it in such a way that the prime values chosen actually meet the condition of prime, and us being sure that it meets the condition stated in the initialisation part of the protocol also. After so much study, we came across the MillerRabin32 function. The MillerRabin32 is used to test the primality of prime values. So when our program chooses two prime numbers, the function checks if they are actually prime and if not the program keeps looping till it get the ones that satisfies the condition. With this, we are sure that the prime values and groups generated are correct.

The values e.g. alpha and beta (α and β) generated after passing it through the hash function (in the protocol) could not be used for subsequent computation. This is because the result from a hash function is in an alphanumeric format. In order to be able to use the value, we had to convert it first to a hexadecimal format before converting it to an integer format. With this, it was able to use the output in other computations.

In order for the data to be passed securely over the internet, we decided to encrypt/decrypt the data using the session keys generated in the protocol. We used AES-CBC128 to securely pass the data over the internet. AES-CBC128 accepts only 128 bit block size and key size. Due to the fact that the session key generated was not up to 128 bit we had to undertake lots of research on how to convert the keys produced into 128 bit so it conforms to the AES-CBC condition. MD5 was used to resolve this problem. MD5 accepts any amount of input and produces a fixed output. The output produced on MD5 is a 128-bit value. With this, we were able to use the 128-bit session key in the AES-CBC encryption/decryption. The session key produced was first passed through MD5 in order to increase the entropy and to be able to use it in AES-CBC. If the starting value has low entropy then the key obtained this way is not too secure, due to this the key obtained had to be slightly increased to make it more secure.

Two prototype applications were built to test if the protocol was realisable.

Overview

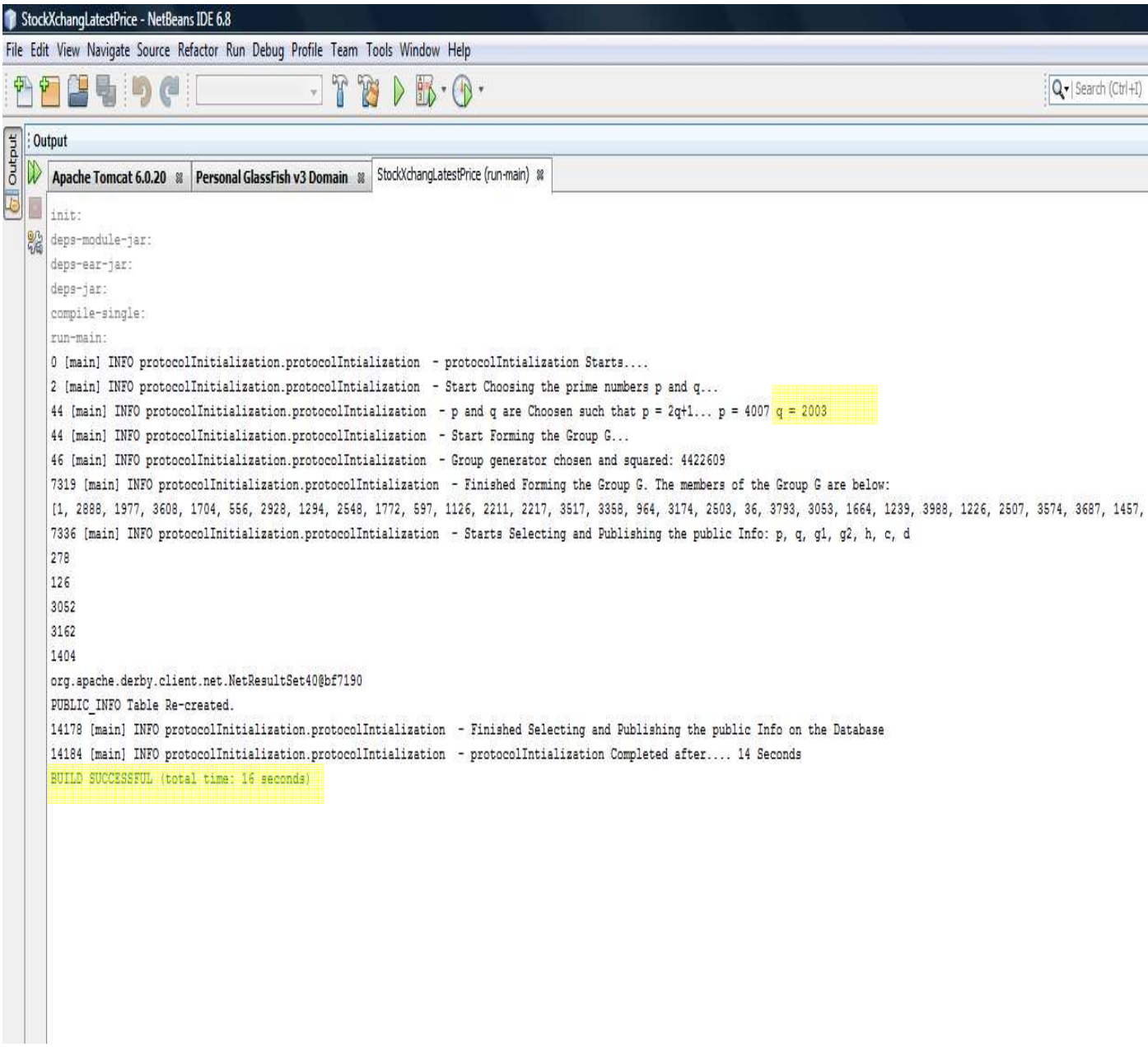
Chapter 3 talks about the protocol specification, the project design and implementation, the problems encountered and how they were resolved. It gave a detailed information on the applications used and why.

Chapter 4: Experimental Result and Critical Evaluation

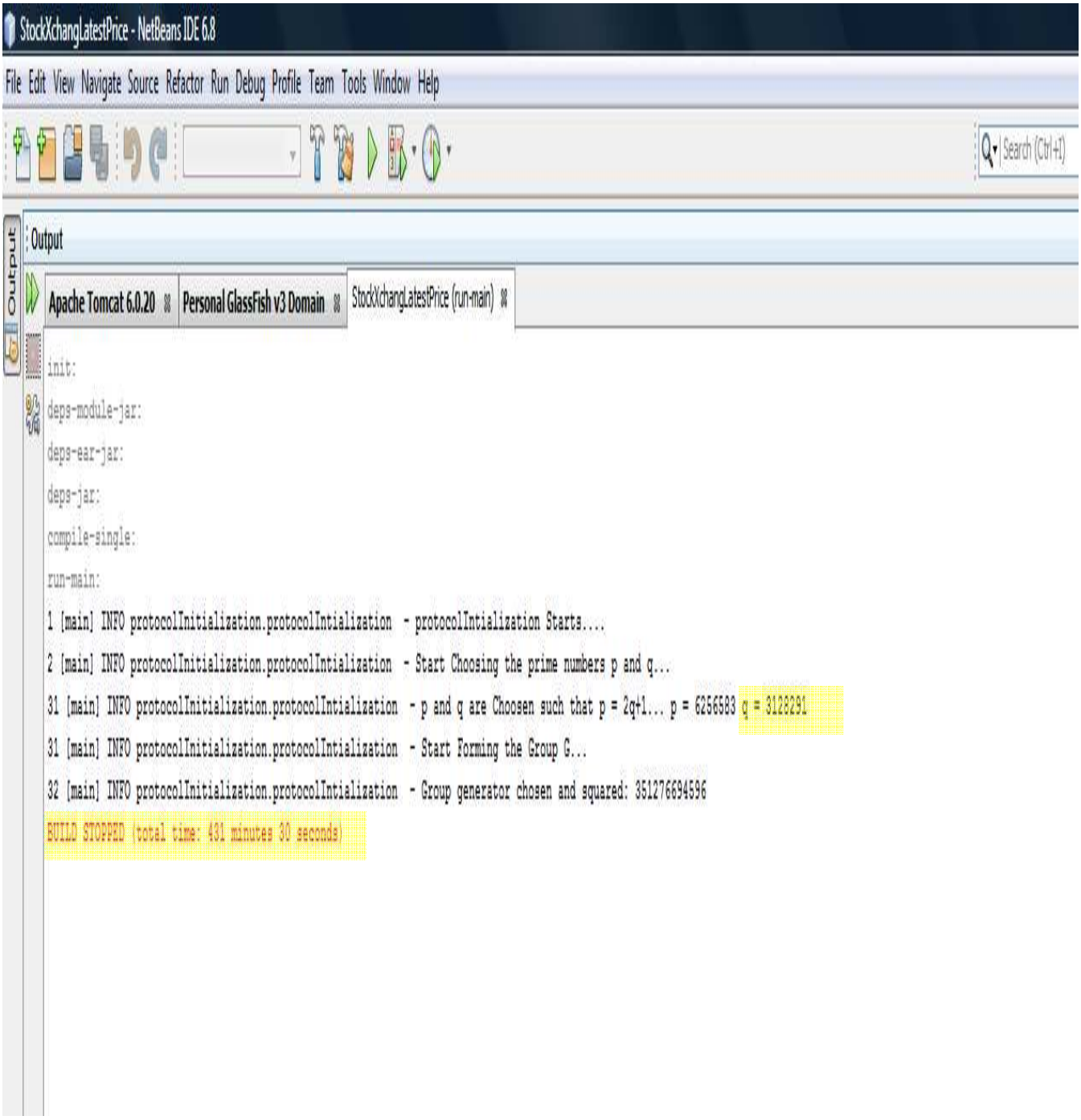
4.1 Experimental Result

This section shows the results obtained during the test carried out. The first result shown is the time it takes to compute the group when a 32-bit and 64-bit prime value was used in the protocol initialisation phase. In JAVA, each digit is represented as 1 byte which is 8 bit;

32-bit prime:

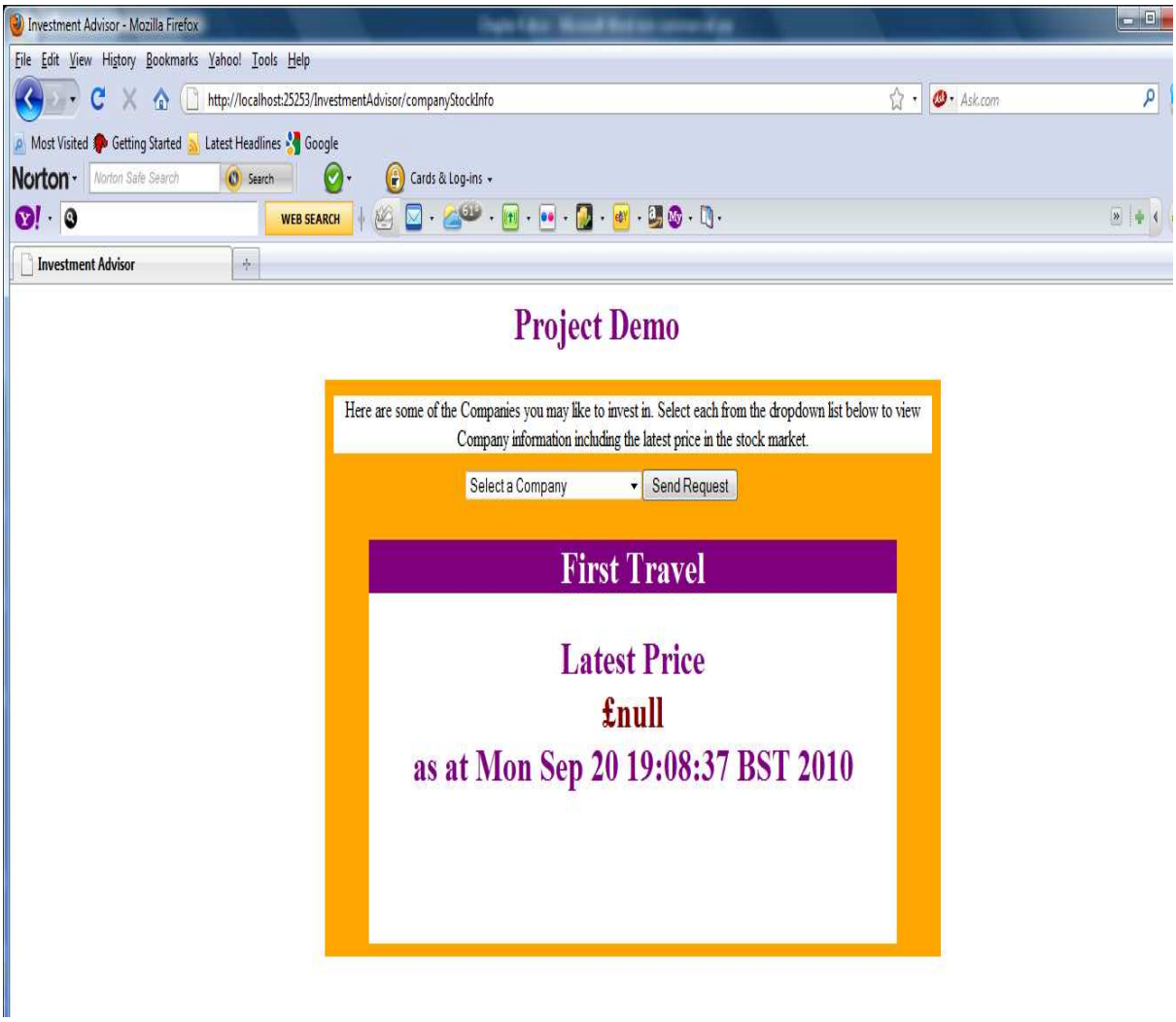


64-bit prime:



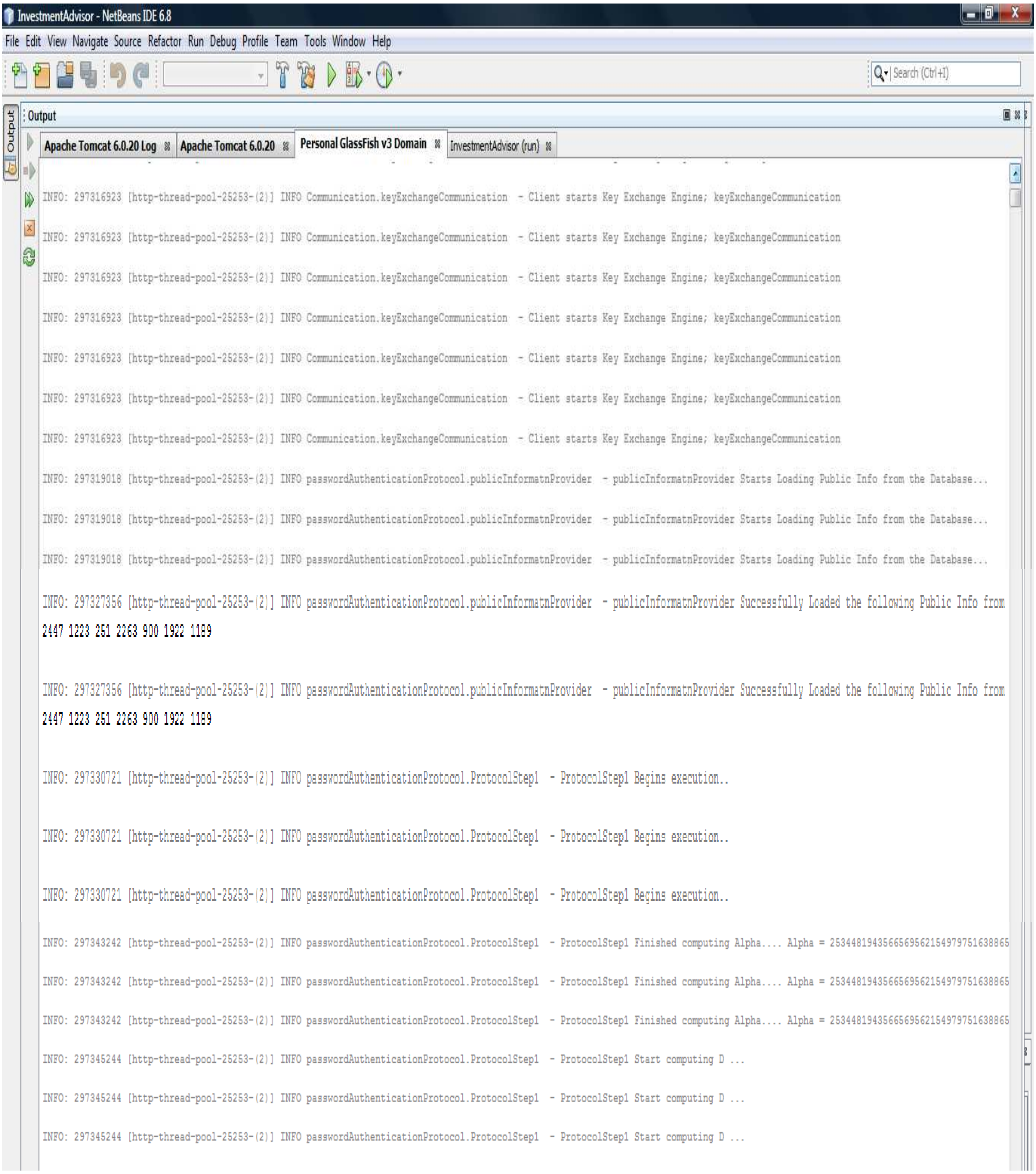
Based on our previous explanation about how the group is formed in (3.3.1), using a 64-bit prime is too overwhelming for the specification of the laptop used. This was the reason why a 32-bit prime was used instead for testing. If deployed to companies with large servers then a 512bit can be used.

The next output shown was the result gotten when the password was changed. It shows the result gotten when the password in the client is different from the password in the server.



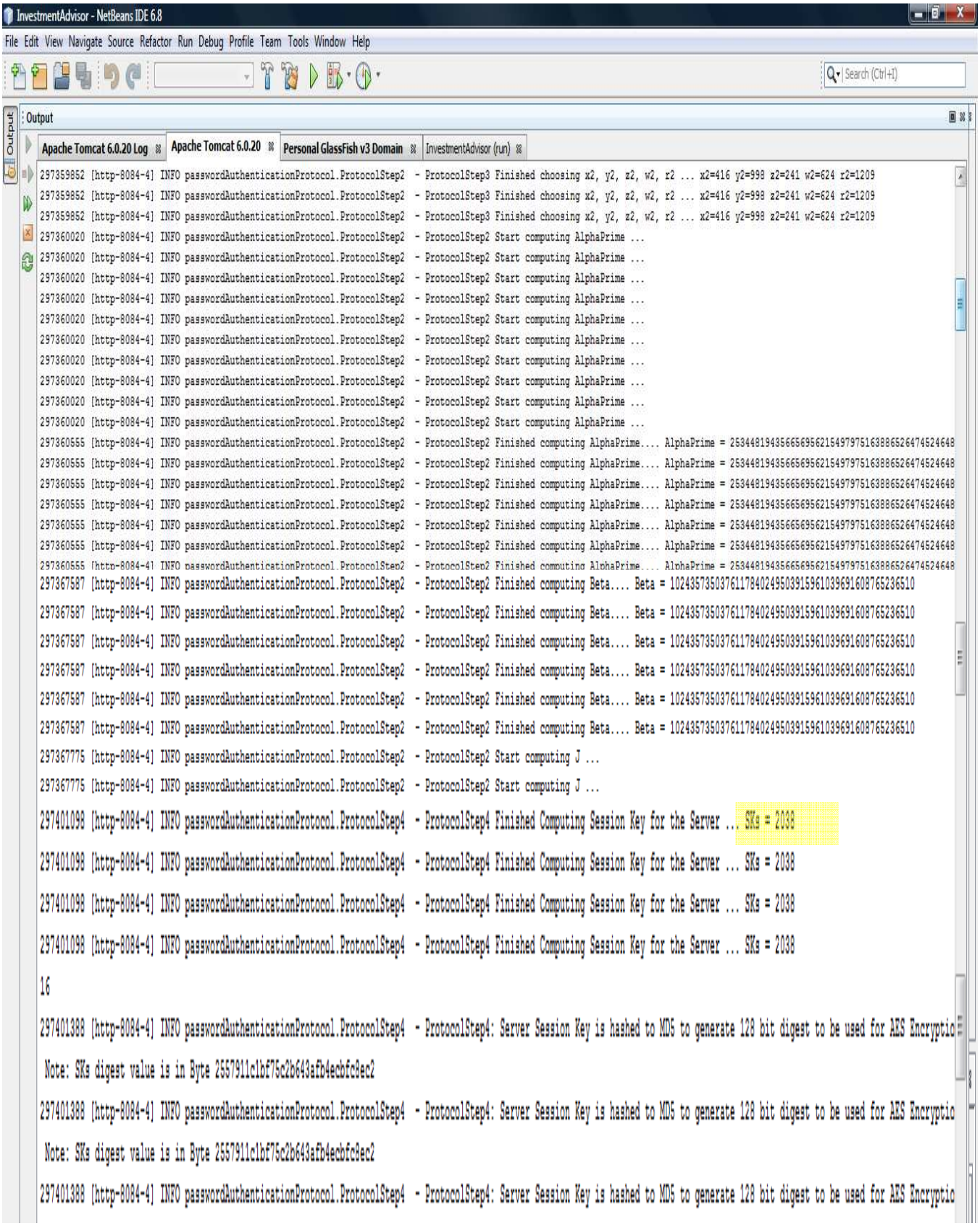
This shows that it is a password-AKE protocol. If the password supplied by the client is different from the one that resides in the server, it returns a null value. With this, an unauthorised company or an adversary would not have access to the services. The session keys generated too would be different.

Client:GlassFish v3



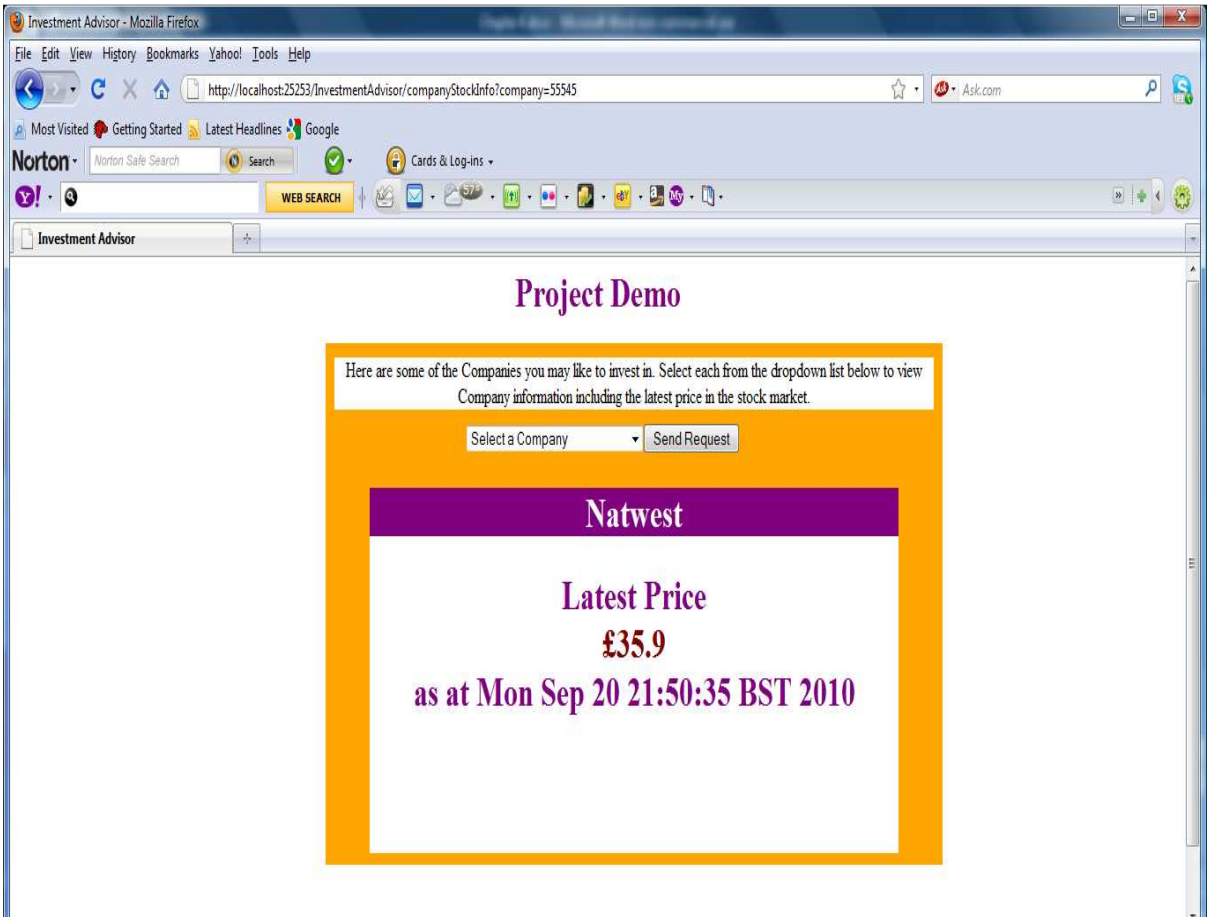
```
INFO: 297348278 [http-thread-pool-25253-(2)] INFO Communication.keyExchangeCommunication  - Sending First message and waiting to recieve Second message

INFO: 297348278 [http-thread-pool-25253-(2)] INFO Communication.keyExchangeCommunication  - Sending First message and waiting to recieve Second message

INFO: Result = [server, 1674, 294, 2230, 2154, 502]
INFO: 297378100 [http-thread-pool-25253-(2)] INFO Communication.keyExchangeCommunication  - Second message RECIEVED!

INFO: 297378100 [http-thread-pool-25253-(2)] INFO Communication.keyExchangeCommunication  - Second message RECIEVED!

INFO: 297386177 [http-thread-pool-25253-(2)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing BetaPrime  ...

INFO: 297386177 [http-thread-pool-25253-(2)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing BetaPrime  ...

INFO: 297389198 [http-thread-pool-25253-(2)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing BetaPrime  ... BetaPrime = 1024357350376117840249503

INFO: 297389198 [http-thread-pool-25253-(2)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing BetaPrime  ... BetaPrime = 1024357350376117840249503

INFO: 297396391 [http-thread-pool-25253-(2)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing Session Key for the Client  ...

INFO: 297396391 [http-thread-pool-25253-(2)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing Session Key for the Client  ...

INFO: 297397396 [http-thread-pool-25253-(2)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing Session Key for the Client ... SKc = 2197

INFO: 297397396 [http-thread-pool-25253-(2)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing Session Key for the Client ... SKc = 2197

INFO: 297397396 [http-thread-pool-25253-(2)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing Session Key for the Client ... SKc = 2197

INFO: 297397396 [http-thread-pool-25253-(2)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing Session Key for the Client ... SKc = 2197

INFO: 297397396 [http-thread-pool-25253-(2)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing Session Key for the Client ... SKc = 2197

INFO: 16
INFO: 297397412 [http-thread-pool-25253-(2)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3: Client Session Key is hashed to MD5 to generate 128 bit digest to be u
 Note: SKc digest value is in Byte ea119a40c1592979f51819b0bd38d39d

INFO: 297397412 [http-thread-pool-25253-(2)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3: Client Session Key is hashed to MD5 to generate 128 bit digest to be u
 Note: SKc digest value is in Byte ea119a40c1592979f51819b0bd38d39d
```

Server: ApacheTomcat 6.0.20

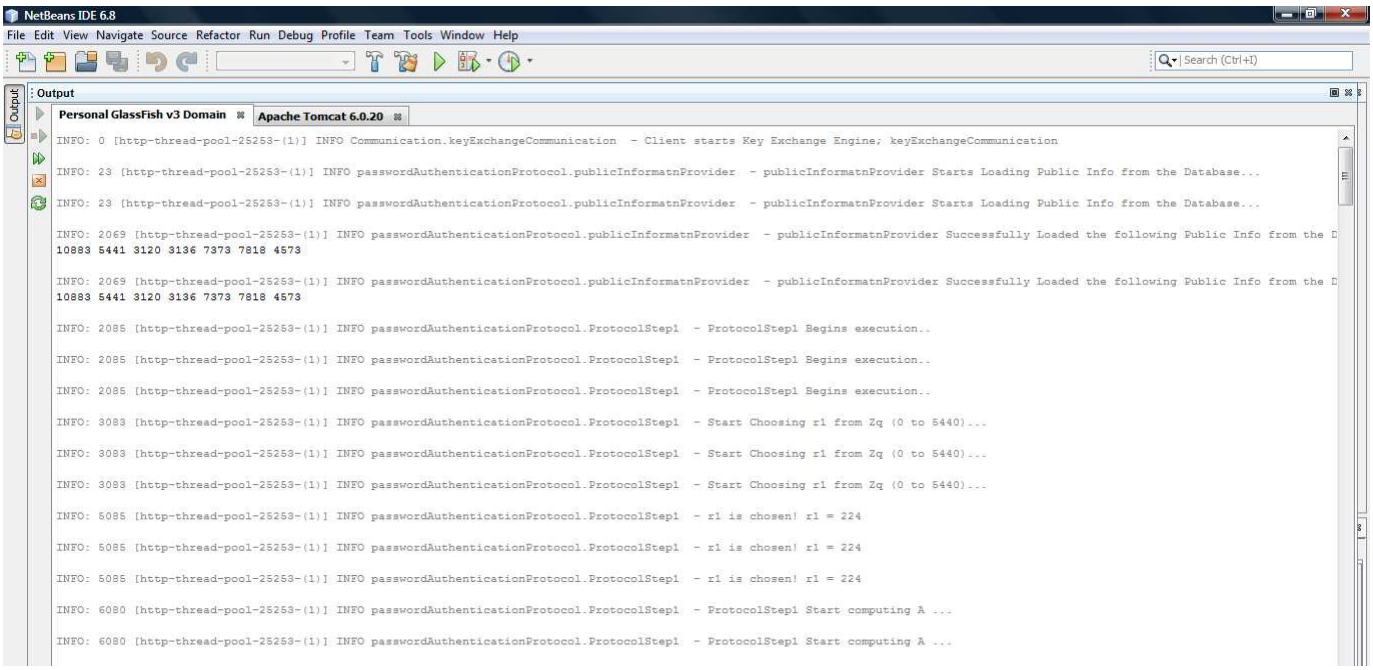


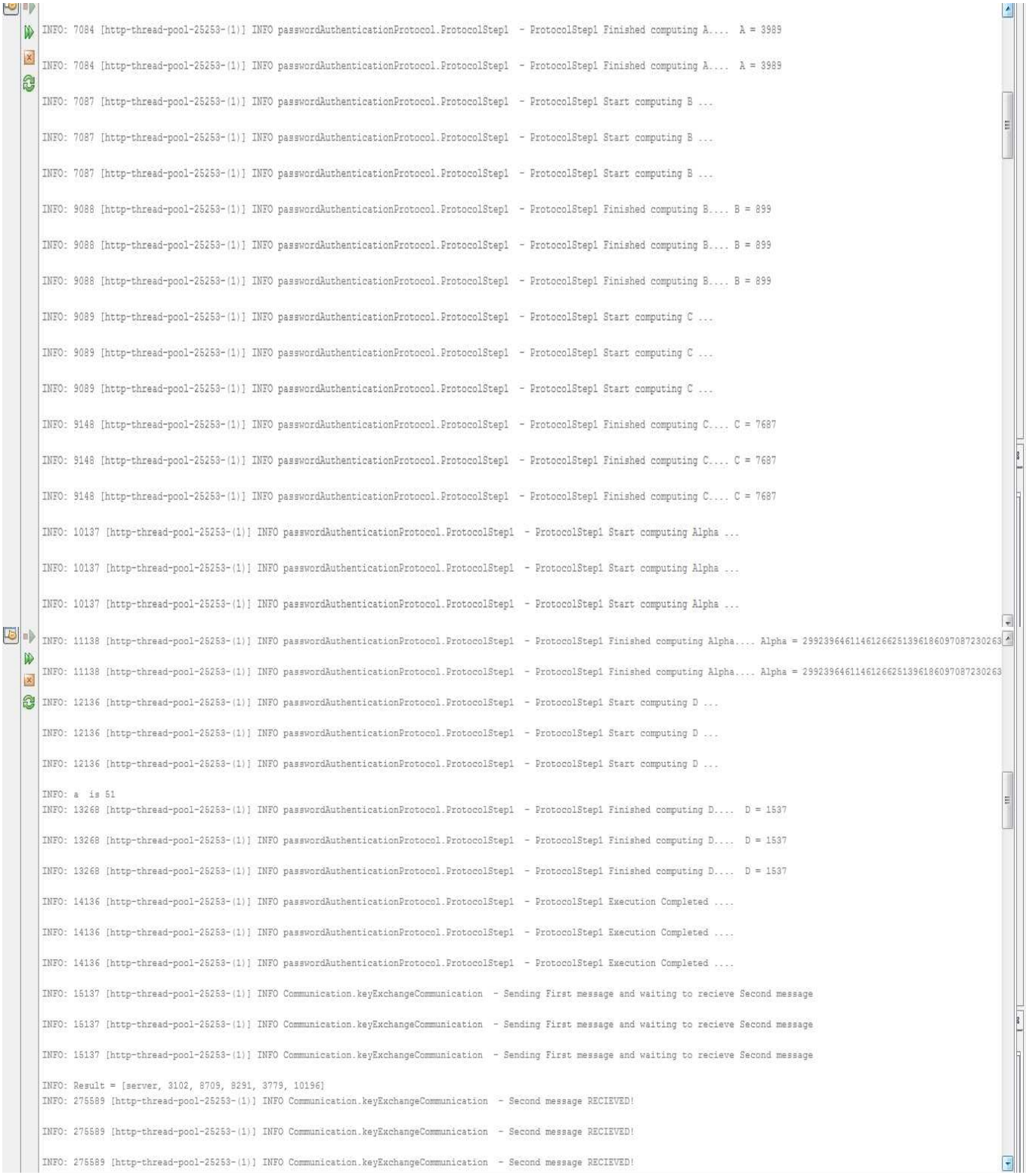
Looking at the session keys, $sk_c = 2197$ and $sk_s = 2038$

The next output show when the passwords are equal.

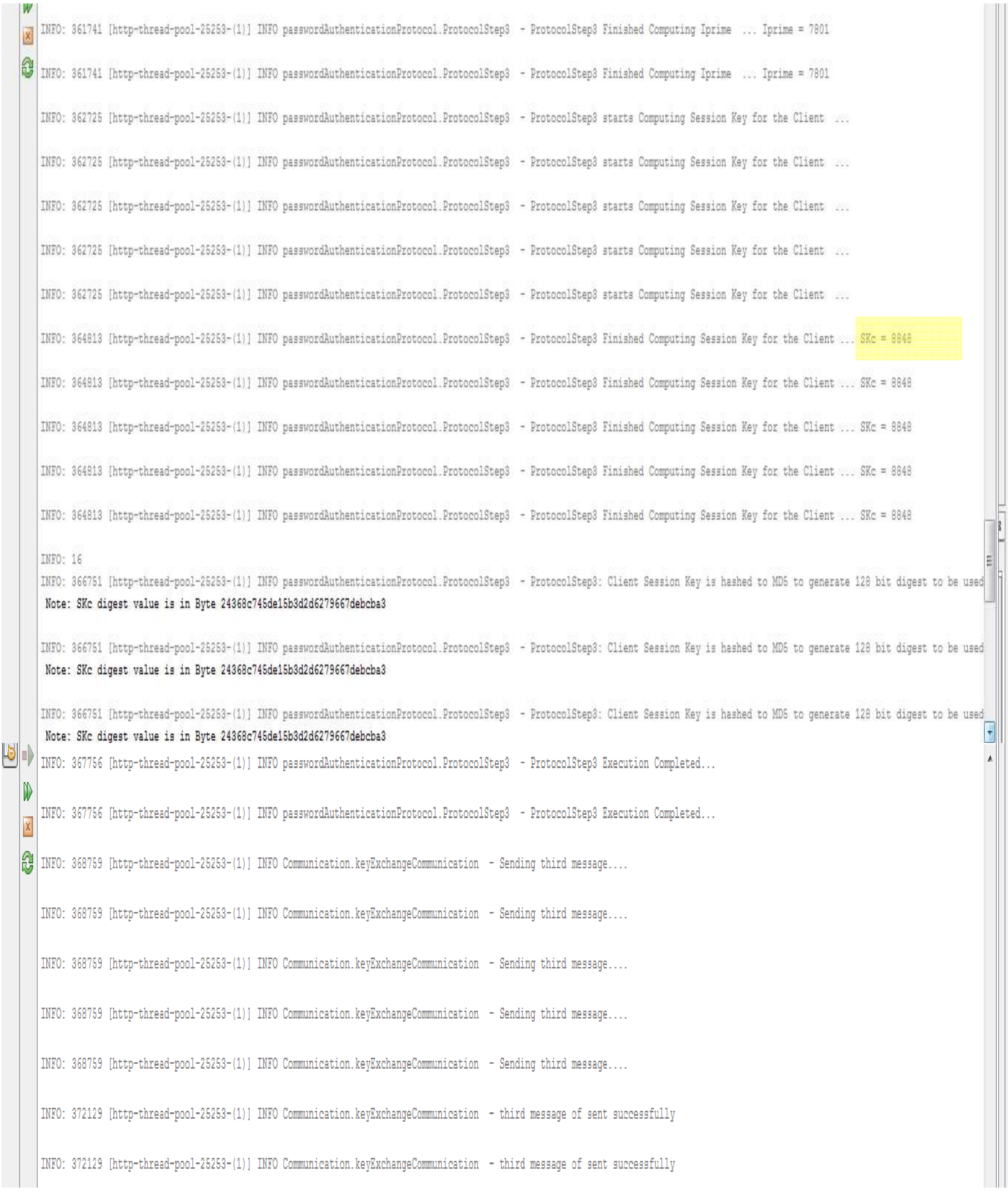


Client:



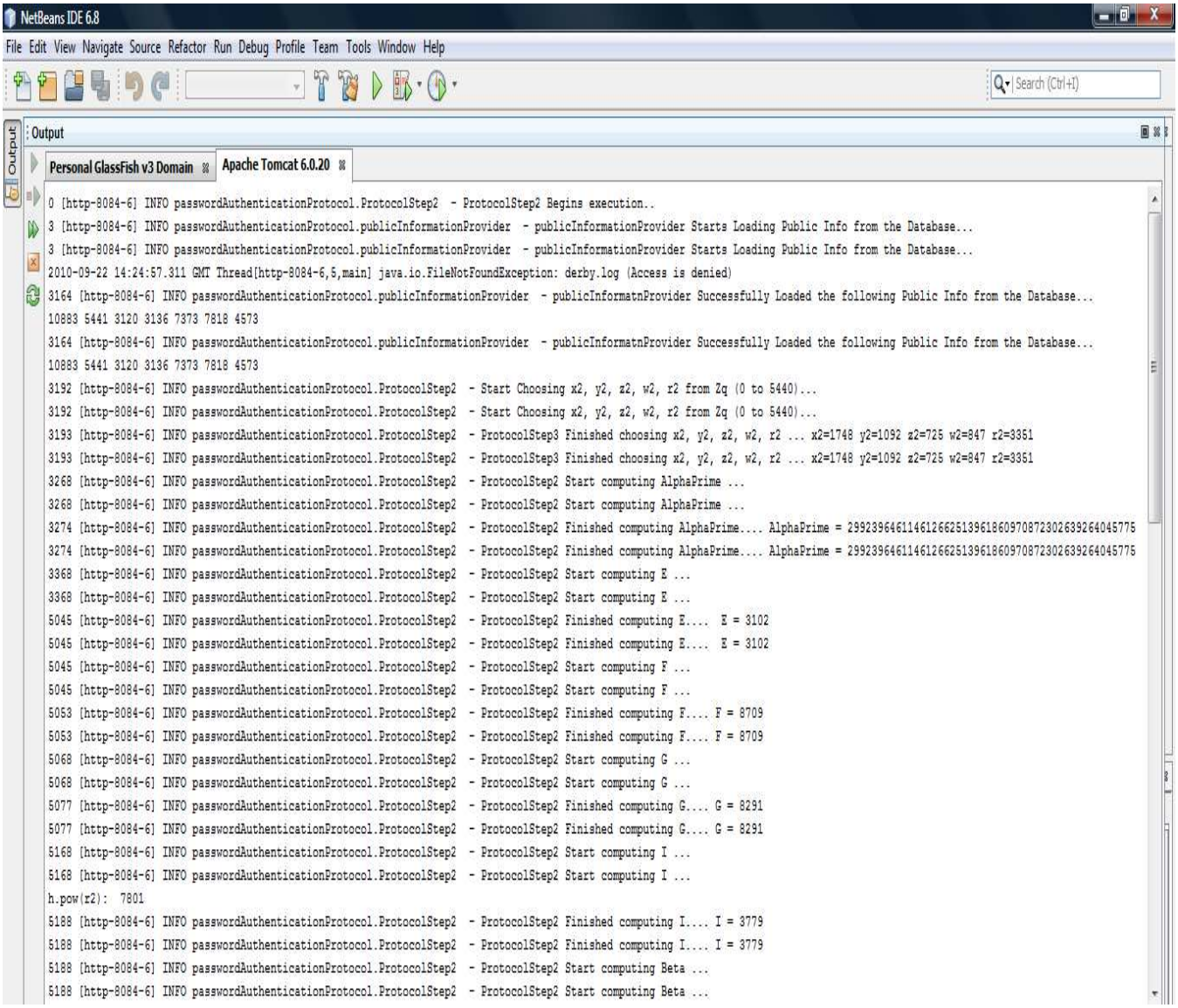

INFO: 7084 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Finished computing A.... A = 3989
INFO: 7084 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Finished computing A.... A = 3989
INFO: 7087 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Start computing B ...
INFO: 7087 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Start computing B ...
INFO: 7087 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Start computing B ...
INFO: 9088 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Finished computing B.... B = 899
INFO: 9088 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Finished computing B.... B = 899
INFO: 9088 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Finished computing B.... B = 899
INFO: 9089 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Start computing C ...
INFO: 9089 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Start computing C ...
INFO: 9089 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Start computing C ...
INFO: 9148 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Finished computing C.... C = 7687
INFO: 9148 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Finished computing C.... C = 7687
INFO: 9148 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Finished computing C.... C = 7687
INFO: 10137 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Start computing Alpha ...
INFO: 10137 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Start computing Alpha ...
INFO: 10137 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Start computing Alpha ...
INFO: 11138 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Finished computing Alpha.... Alpha = 29923964611461266251396186097087230263
INFO: 11138 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Finished computing Alpha.... Alpha = 29923964611461266251396186097087230263
INFO: 12136 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Start computing D ...
INFO: 12136 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Start computing D ...
INFO: 12136 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Start computing D ...
INFO: a is 51
INFO: 13268 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Finished computing D.... D = 1537
INFO: 13268 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Finished computing D.... D = 1537
INFO: 13268 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Finished computing D.... D = 1537
INFO: 14136 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Execution Completed ....
INFO: 14136 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Execution Completed ....
INFO: 14136 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep1 - ProtocolStep1 Execution Completed ....
INFO: 15137 [http-thread-pool-25253-(1)] INFO Communication.keyExchangeCommunication - Sending First message and waiting to recieve Second message
INFO: 15137 [http-thread-pool-25253-(1)] INFO Communication.keyExchangeCommunication - Sending First message and waiting to recieve Second message
INFO: 15137 [http-thread-pool-25253-(1)] INFO Communication.keyExchangeCommunication - Sending First message and waiting to recieve Second message
INFO: Result = [server, 3102, 8709, 8291, 3779, 10196]
INFO: 275589 [http-thread-pool-25253-(1)] INFO Communication.keyExchangeCommunication - Second message RECIEVED!
INFO: 275589 [http-thread-pool-25253-(1)] INFO Communication.keyExchangeCommunication - Second message RECIEVED!
INFO: 275589 [http-thread-pool-25253-(1)] INFO Communication.keyExchangeCommunication - Second message RECIEVED!

```
INFO: 281599 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Begins execution...
INFO: 281599 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Begins execution...
INFO: 281599 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Begins execution...
INFO: 283590 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts choosing x1, y1, z1, w1 from Zq (0 to 5440)...
INFO: 283590 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts choosing x1, y1, z1, w1 from Zq (0 to 5440)...
INFO: 283590 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts choosing x1, y1, z1, w1 from Zq (0 to 5440)...
INFO: 283590 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts choosing x1, y1, z1, w1 from Zq (0 to 5440)...
INFO: 283590 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts choosing x1, y1, z1, w1 from Zq (0 to 5440)...
INFO: 285591 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished choosing x1, y1, z1, w1 ... x1=3823 y1=4819 z1=3255 w1=1695
INFO: 285591 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished choosing x1, y1, z1, w1 ... x1=3823 y1=4819 z1=3255 w1=1695
INFO: 285591 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished choosing x1, y1, z1, w1 ... x1=3823 y1=4819 z1=3255 w1=1695
INFO: 285591 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished choosing x1, y1, z1, w1 ... x1=3823 y1=4819 z1=3255 w1=1695
INFO: 285591 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished choosing x1, y1, z1, w1 ... x1=3823 y1=4819 z1=3255 w1=1695
INFO: 287591 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing BetaPrime  ...
INFO: 287591 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing BetaPrime  ...
INFO: 287591 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing BetaPrime  ...
INFO: 287591 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing BetaPrime  ...
INFO: 290591 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing BetaPrime  ... BetaPrime = 149005966875434259535804628
INFO: 290591 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing BetaPrime  ... BetaPrime = 149005966875434259535804628
INFO: 290591 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing BetaPrime  ... BetaPrime = 149005966875434259535804628
INFO: 290591 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing BetaPrime  ... BetaPrime = 149005966875434259535804628
INFO: 290591 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing BetaPrime  ... BetaPrime = 149005966875434259535804628
INFO: 292592 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing K ...
INFO: 292592 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing K ...
INFO: 292592 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing K ...
INFO: 292592 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing K ...
INFO: 292592 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing K ...
INFO: 358715 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing K ... K = 260
INFO: 358715 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing K ... K = 260
INFO: 358715 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing K ... K = 260
INFO: 358715 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing K ... K = 260
INFO: 358715 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing K ... K = 260
INFO: 359716 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing Iprime  ...
```



```
INFO: 361741 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing Iprime  ... Iprime = 7801
INFO: 361741 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing Iprime  ... Iprime = 7801
INFO: 362725 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing Session Key for the Client  ...
INFO: 362725 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing Session Key for the Client  ...
INFO: 362725 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing Session Key for the Client  ...
INFO: 362725 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing Session Key for the Client  ...
INFO: 362725 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 starts Computing Session Key for the Client  ...
INFO: 364813 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing Session Key for the Client ... SKc = 8848
INFO: 364813 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing Session Key for the Client ... SKc = 8848
INFO: 364813 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing Session Key for the Client ... SKc = 8848
INFO: 364813 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing Session Key for the Client ... SKc = 8848
INFO: 364813 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Finished Computing Session Key for the Client ... SKc = 8848
INFO: 16
INFO: 366751 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3: Client Session Key is hashed to MD5 to generate 128 bit digest to be used
 Note: SKc digest value is in Byte 24368c745de15b3d2d6279667debcba3
INFO: 366751 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3: Client Session Key is hashed to MD5 to generate 128 bit digest to be used
 Note: SKc digest value is in Byte 24368c745de15b3d2d6279667debcba3
INFO: 366751 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3: Client Session Key is hashed to MD5 to generate 128 bit digest to be used
 Note: SKc digest value is in Byte 24368c745de15b3d2d6279667debcba3
INFO: 367756 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Execution Completed...
INFO: 367756 [http-thread-pool-25253-(1)] INFO passwordAuthenticationProtocol.ProtocolStep3  - ProtocolStep3 Execution Completed...
INFO: 368759 [http-thread-pool-25253-(1)] INFO Communication.keyExchangeCommunication  - Sending third message....
INFO: 368759 [http-thread-pool-25253-(1)] INFO Communication.keyExchangeCommunication  - Sending third message....
INFO: 368759 [http-thread-pool-25253-(1)] INFO Communication.keyExchangeCommunication  - Sending third message....
INFO: 368759 [http-thread-pool-25253-(1)] INFO Communication.keyExchangeCommunication  - Sending third message....
INFO: 368759 [http-thread-pool-25253-(1)] INFO Communication.keyExchangeCommunication  - Sending third message....
INFO: 372129 [http-thread-pool-25253-(1)] INFO Communication.keyExchangeCommunication  - third message of sent successfully
INFO: 372129 [http-thread-pool-25253-(1)] INFO Communication.keyExchangeCommunication  - third message of sent successfully
```

Sever


NetBeans IDE 6.8
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
Output
Personal GlassFish v3 Domain
Apache Tomcat 6.0.20
0 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Begins execution..
3 [http-8084-6] INFO passwordAuthenticationProtocol.publicInformationProvider - publicInformationProvider Starts Loading Public Info from the Database...
3 [http-8084-6] INFO passwordAuthenticationProtocol.publicInformationProvider - publicInformationProvider Starts Loading Public Info from the Database...
2010-09-22 14:24:57.311 GMT Thread[http-8084-6,5,main] java.io.FileNotFoundException: derby.log (Access is denied)
3164 [http-8084-6] INFO passwordAuthenticationProtocol.publicInformationProvider - publicInformatnProvider Successfully Loaded the following Public Info from the Database...
10883 5441 3120 3136 7373 7818 4573
3164 [http-8084-6] INFO passwordAuthenticationProtocol.publicInformationProvider - publicInformatnProvider Successfully Loaded the following Public Info from the Database...
10883 5441 3120 3136 7373 7818 4573
3192 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - Start Choosing x2, y2, z2, w2, r2 from Zq (0 to 5440)...
3192 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - Start Choosing x2, y2, z2, w2, r2 from Zq (0 to 5440)...
3193 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep3 Finished choosing x2, y2, z2, w2, r2 ... x2=1748 y2=1092 z2=725 w2=847 r2=3351
3193 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep3 Finished choosing x2, y2, z2, w2, r2 ... x2=1748 y2=1092 z2=725 w2=847 r2=3351
3268 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Start computing AlphaPrime ...
3268 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Start computing AlphaPrime ...
3274 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Finished computing AlphaPrime.... AlphaPrime = 29923964611461266251396186097087230263926404577 5
3274 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Finished computing AlphaPrime.... AlphaPrime = 29923964611461266251396186097087230263926404577 5
3368 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Start computing E ...
3368 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Start computing E ...
5045 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Finished computing E.... E = 3102
5045 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Finished computing E.... E = 3102
5045 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Start computing F ...
5045 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Start computing F ...
5053 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Finished computing F.... F = 8709
5053 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Finished computing F.... F = 8709
5068 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Start computing G ...
5068 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Start computing G ...
5077 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Finished computing G.... G = 8291
5077 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Finished computing G.... G = 8291
5168 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Start computing I ...
5168 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Start computing I ...
h.pow(r2): 7801
5188 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Finished computing I.... I = 3779
5188 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Finished computing I.... I = 3779
5188 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Start computing Beta ...
5188 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2 - ProtocolStep2 Start computing Beta ...

```
5053 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Finished computing F.... F = 8709
5053 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Finished computing F.... F = 8709
5068 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Start computing G ...
5068 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Start computing G ...
5077 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Finished computing G.... G = 8291
5077 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Finished computing G.... G = 8291
5168 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Start computing I ...
5168 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Start computing I ...
h.pow(r2):  7801
5188 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Finished computing I.... I = 3779
5188 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Finished computing I.... I = 3779
5188 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Start computing Beta ...
5188 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Start computing Beta ...
5271 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Finished computing Beta.... Beta = 1490059668754342595358046289316924762049851563334
5271 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Finished computing Beta.... Beta = 1490059668754342595358046289316924762049851563334
5271 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Start computing J ...
5271 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Start computing J ...
256889 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Finished computing J....  J = 10196
256889 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Finished computing J....  J = 10196
256889 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Execution Completed ....
256889 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep2  - ProtocolStep2 Execution Completed ....
353392 [http-8084-6] INFO passwordAuthenticationProtocol.publicInformationProvider  - publicInformationProvider Starts Loading Public Info from the Database...
353392 [http-8084-6] INFO passwordAuthenticationProtocol.publicInformationProvider  - publicInformationProvider Starts Loading Public Info from the Database...
353392 [http-8084-6] INFO passwordAuthenticationProtocol.publicInformationProvider  - publicInformationProvider Starts Loading Public Info from the Database...
353407 [http-8084-6] INFO passwordAuthenticationProtocol.publicInformationProvider  - publicInformatnProvider Successfully Loaded the following Public Info from the Database...
10883 5441 3120 3136 7373 7818 4573
353407 [http-8084-6] INFO passwordAuthenticationProtocol.publicInformationProvider  - publicInformatnProvider Successfully Loaded the following Public Info from the Database...
10883 5441 3120 3136 7373 7818 4573
353407 [http-8084-6] INFO passwordAuthenticationProtocol.publicInformationProvider  - publicInformatnProvider Successfully Loaded the following Public Info from the Database...
10883 5441 3120 3136 7373 7818 4573
353450 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Begins execution...
353450 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Begins execution...
353450 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Begins execution...
353450 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Begins execution...
353450 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 starts Computing Cprime  ...
```

```
353450 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Begins execution...
353450 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Begins execution...
353450 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Begins execution...
353450 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Begins execution...
353450 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 starts Computing Cprime  ...
353450 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 starts Computing Cprime  ...
353450 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 starts Computing Cprime  ...
353450 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 starts Computing Cprime  ...
353545 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Finished Computing Cprime  ... Cprime = 3626
353545 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Finished Computing Cprime  ... Cprime = 3626
353545 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Finished Computing Cprime  ... Cprime = 3626
353545 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Finished Computing Cprime  ... Cprime = 3626
353545 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 starts Computing Session Key for the Server  ...
353545 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 starts Computing Session Key for the Server  ...
353545 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 starts Computing Session Key for the Server  ...
353545 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 starts Computing Session Key for the Server  ...
353553 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Finished Computing Session Key for the Server ... SKs = 8848
353553 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Finished Computing Session Key for the Server ... SKs = 8848
353553 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Finished Computing Session Key for the Server ... SKs = 8848
353553 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Finished Computing Session Key for the Server ... SKs = 8848
16
353555 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4: Server Session Key is hashed to MD5 to generate 128 bit digest to be used for AES Encryption/D
 Note: SKs digest value is in Byte 24368c745de15b3d2d6279667debcba3
353555 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4: Server Session Key is hashed to MD5 to generate 128 bit digest to be used for AES Encryption/D
 Note: SKs digest value is in Byte 24368c745de15b3d2d6279667debcba3
353555 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4: Server Session Key is hashed to MD5 to generate 128 bit digest to be used for AES Encryption/D
 Note: SKs digest value is in Byte 24368c745de15b3d2d6279667debcba3
353555 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4: Server Session Key is hashed to MD5 to generate 128 bit digest to be used for AES Encryption/D
 Note: SKs digest value is in Byte 24368c745de15b3d2d6279667debcba3
353556 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Execution Completed...
353556 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Execution Completed...
353556 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Execution Completed...
353556 [http-8084-6] INFO passwordAuthenticationProtocol.ProtocolStep4  - ProtocolStep4 Execution Completed...
```

Looking at the session keys sk_c = 8848 and sk_s = 8848.

Each digit represents a byte. 1 byte = 8 bits

So 4 digits = 32 bits. As I said earlier, this was used because of the specification of the laptop.

4.2 Critical Evaluation

The protocol achieves key exchange not mutual authentication [3]. Key exchange is the process of having equal session keys at both ends. To achieve equal session keys, what is really needed is the public information, password, multiplication in the group operation, division in the group operation and the exchange that takes place. The protocol already has public parameters and password which is passed from the server to the client and shared among them. An adversary cannot possibly know the parameters used if they intercept the password, so there is no way they can forge as a valid participant. The sign and verify operation is too fast compared to the exponentiation operation on the protocol. Since this is a onetime signature (not a full signature), signing/verifying is very fast, hence not crucially important for the efficiency considerations.
Besides it does not have any significant effect towards achieving equal session key, which is the main aim of the protocol.

PKI requires the key technology (public and private key) which involves using one key to encrypt and the other to decrypt. These keys have some issues attributed to it such as key revocation, key management and deleting the revocated key from the server. Key revocation is the process of cancelling an expired key. Imagine a scenario whereby a private key has been revoked but the public key is still active; it causes some sort of confusion when each party wants to communicate. This and many more reasons were the motivation behind examining another technique which is password based.

In PAKE, trust is built on the password while in PKI; trust is built on the trusted third party (Certification Authority). If the CA for PKI has been tampered with then the participants would have no choice than to accept what it certifies even if it has been compromised.

In PAKE two parties are certified while in PKI only one party is certified. The two parties in PAKE must have a password which is shared among them and if the password is not the same then the requester is not granted the information it needs. While in PKI, only the requester is certified by the CA before it has access to the server.

The protocol is relatively slow because it involves a lot of exponentiation operation. Exponentiation simply means repeated multiplication of the base with the exponent. It is written as g^x where g is the base and x is the exponent. So if x=150 and g= 60, it simply means we should multiply 60 in 150 times (60^{150}). This of course is a time consuming operation. Imagine the amount of times this operation appears in the protocol before it finally generates the session key. So if a request is made on the client, it would have to wait for all this computation to take place before it eventually gets the data it requires.

A protocol is supposed to be flexible. It shouldn't require a particular number of participants (client and server). After a new client subscribes to the provider of these services, he should be given a password or something which enables him to join the network almost immediately but this doesn't apply in the case of this protocol. The protocol is not flexible. It requires a fixed number of participants [3]. From the material " we have a fixed set of protocol participants (principals) each of which is either client $C \in$ Client or a server $S \in$ Server (Client and Server are disjoint). We let User $\stackrel{\text{def}}{=}$ Client $\cup$ Server". The clients and server that can use this protocol are fixed. This means that client can't just join without going through lots of processes. It would have to be given the password and public parameters first, then the whole computation would have to take place again and the server would have to readjust its database in order to fix the new member in.

Although the protocol is relatively slow, it is not as expensive as PKI because it wouldn't require a third party and does not have the key related issues.

Chapter Five: Conclusion and Further Work

5.1 Conclusion

The Password-Authenticated Key Exchange proposed by J. Katz, R. Ostrovsky and M. Yung has been implemented in this project. It was tested using two applications and the exchange of messages was achieved using Web Service. The objective was to implement a secure channel based on PAKE as opposed to PKI which currently exists on the internet and to ensure that messages are passed securely without any adversary having access to the data that is transferred. It also prevents the issues attributed to PKI and eliminates the cost of a trusted third party.

In order to achieve this, a couple of research and implementation projects were put in place. Chapter 2 talks about the background research of different PAKE protocols, their strength and weaknesses. Chapter 3 explains the design considerations, implementation details, problems encountered and how it was tackled. In section 3.3 we showed the two applications that were used to test the protocol and why they were used. Section 3.3.1 shows a detailed description of how each value in the protocol was computed. Finally section 3.3.2 gives the applications that were implemented at the server and the client, and how all these applications work in order to achieve the overall outcome of the project.

For the client and server to communicate they must share the same public information and password. If the password at the client is different from the one residing in the server, a null value is returned when a request is made. This ensures that an adversary cannot get information since it does not have access to the password. If it does, it still does not have access to the parameters used to compute the public information. So in a way it cannot have access to the information it needs from the server.

The password is embedded in the protocol and secured, therefore it cannot be intercepted. It does not require a client to type the password on the browser which is the method adopted in PKI. The two participants are authenticated and the trust is based on a password unlike PKI where only one party is certified before sending the request to the server.

A 512–bit prime values can be used and makes the protocol even more secure because an adversary cannot make such a continuous search on the prime value, but it wasn't used here. This is because of the amount of time it was taking the computer to form the group using such large primes and the specification of the laptop which was discussed in section 3.3.2.1. Due to this reason, a 32-bit prime was used to perform the test. If the protocol is deployed to a mainframe system or system with large processor and system type, a 512-bit can be used and shouldn't encounter any problem in computation. It should also be relatively fast.

This protocol can be used in companies where security is their focus.

5.2 Further Work

The protocol can be improved by adding mutual authentication to it. This can be achieved by adding a fourth message. Mutual authentication is the process whereby both parties prove that they have the same password.

Reference:

[1] Goldreich O., Lindell Y. Session Key Generation Using Human Passwords Only. Personal Communication and Crypto 2000 Rump Session, pg 1-91, 2005. Available at http://eprint.iacr.org/2000/057

[2] Marchesini J., Smith S. Modelling Public Key Infrastructure in the Real World. 2nd European PKI Workshop: Research and Applications, LNCS 3545, pg 1-17, 2005

[3] Katz J., Ostrovsky R., Yung M. Efficient Password-Authenticated Key Exchange Using Human Memorable Passwords. Lecture Notes in Computer Science; Advances in Cryptology Eurocrypt 2001, Vol. 2045, Pg475 – 494, 2001

[4] Diffie W., Hellman M. E. New Directions in Cryptography. *IEEE Trans. on Info. Theory*, Vol. IT-22, pg. 644-654, 1976

[5] http://www.entrust.com/corporate/history.htm

[6] Jens-Peter Kaps. Public Key Infrastructure. Pg 1-7, 1997

[7] Netscape. *The SSL (Secure Sockets Layer) 3.0 Protocol*. Netscape Communications Corp, 1995

[8] Han J.H., Kim Y.J., Jun S.I., Chung K.I., Seo C.H. Implementation of ECC/ECDSA cryptography algorithms based on Java card. Proceedings of 22nd IEEE International Conference on Distributed Computing Systems, pg.272-276, 2002.

[9] Hankerson D., Menezes A., Vanstone S. Guide to Elliptic Curve Cryptography. Springer-Verlag, New York, pg 1-332, 2004

[10] NetScaler 2048-bit SSL Performance. Citrix community, pg 1-8
July 2010 available at:
http://www.citrix.com/site/resources/dynamic/salesdocs/Citrix_NS_MPX_SSL_2048-bit.pdf

[11] Toorani M., Shirazi A. A. B. LPKI - A Lightweight Public Key Infrastructure for the Mobile Environments. IEEE International Conference on Communication Systems (IEEE ICCS'08), pg.162-166, 2008. Available at http://arxiv.org/list/cs.CR/1002

[12] Charles Dollars. PKI Management and Archive Issues. ECURE Conference, Pg 1-24, 2002, available at http://www.asu.edu/ecure/2002/dollar/index.html

[13] Lekkas D., Gritzalis D. E-Passports as a Means Towards the First World-Wide Public Key Infrastructure. EuroPKI 2007, LNCS 4582, pg. 34–48, 2007.
© Springer-Verlag Berlin Heidelberg 2007

[14] Bellovin S.M, Merritt M. Augmented Encrypted Key Exchange: A Password-based protocol secure against dictionary attacks and password -file compromise. ACM Conference on Computer and Communications Security, pg. 244-250, 1993

[15]Elgamal T. A public-key cryptosystem and a signature scheme based on discrete logarithms. IEEE Trans. Information Theory, vol.IT-31, no.4, pg.469-472, 1985

[16] Bellovin S.M., Merritt M. Encrypted Key Exchange: password-based protocols secure against dictionary attacks. In Proc. IEEE Comp. Society Symp. on Research in Security and Privacy, pg. 72-84, 1992

[17] Craig G., Philip M., Zulfikar R. Password Authenticated Key Exchange Using Hidden Smooth Subgroups. ACM Conference on Computer and Communication Security, pg 299-309, 2005

[18] Nyberg K., Rueppel R.A., Message Recovery for Signature Scheme Based on the Discrete Logarithm Problem. Eurocrypt 94, pp. 182-193, 1994

[19]Patel S. Number Theoretic Attacks on Secure Password Schemes. IEEE Symposium on Security and Privacy, pg 23
6 -247, 1997

[20] Bellare M., Rogaway P. The AuthA Protocol for Password Based Authenticated Key Exchange. Contribution to IEEE P1363, pg 1-7, 2000

[21] Kwon T. Ultimate Solution to Authentication via Memorable Password. Contribution to the IEEE P1363 study group for Future PKC Standards, pg 1-22, 2000

[22] Cramer R., Shoup V. A practical public key cryptosystem provably secure against ciphertext attack. Crypto 98 Lecture Notes in Computer Science, Vol 1462, pg 13-25, 1998

[23]Jablon D. Strong password-only authenticated key exchange. ACM SIGCOMM Computer Comm. Review, vol.26, pp.5-26, 1996

[24] Jablon D. Extended Password Key Exchange Protocols Immune to Dictionary Attack. Proceedings Sixth IEEE workshops, pg 248 – 255, 1997

[25]Kwon T., Song J. Secure Agreement scheme for g^{xy} via password authentication. IEEE Electronics Letters, vol.35, pp.892-893, 1999

[26] Boyko V., MacKenzie P., Patel S. Probably Secure Password Authenticated Key Exchange Using Diffie -Hellman. Advances in Cryptology Eurocrypt 00, LNCS Vol 1807, pg 156-171, Springer-Verlag 2000

[27] Even S., Goldreich O., Micali S. On-line/Off-line Digital Signatures. Journals of Cryptology, vol 9, pg 35–67, 1996

[28] Wu T. The Secure Remote Password Protocol. Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium, pg. 97-111, 1998

[29] MacKenzie P., Swaminathan R. Secure Network Authentication with Password Identification. Presented to IEEE P1363a, pg 1-11, 1999

[30] Boneh D. The Decision Diffie-Hellman Problem. In Proceedings of the *Third Algorithmic Number Theory Symposium*, Lecture Notes in Computer Science, Vol. 1423, Springer-Verlag, pg. 48-63, 1998

[31] Bellare M., Canetti R., Krawczyk H. A Modular Approach to the Design and Analysis of Authentication and Key Exchange Protocols. Annual ACM Symposium on Theory of Computing, pg 419 - 428, 1998

[32] Naor M., Yung M. Universal One-Way Hash Functions and Their Cryptographic Applications. Annual ACM Symposium on Theory of Computing: Proceedings of the twenty-first annual ACM symposium on Theory of computing, pg 33-43, 1989

[33] Bellare M., Pointcheaval D., Rogaway P. Authenticated Key Exchange Secure Against Dictionary Attacks. Advances in Cryptology Eurocrypt'00, LNCS Vol. 1807. pg 139-155, Springer-Verlag 2000

[34] Ellison C., Schneier B. Ten Risks of PKI: What you're not being told about Public Key Infrastructure. Computer Security Journal, Volume XVI, Number 1, pg 1-8, 2000

[35] Ellison C., Frantz B., Lampson B., Rivest R., Thomas B., Ylonen T. SPKI Certificate Theory. IETF RFC 2693, pg 1-38, 1999

[36] Ellison C. Improvements on Conventional PKI Wisdom. In Proceedings of the 1st Annual PKI Research Workshop, pg 2-219, 2002

[37] Farrell S., Housley R. An Internet Attribute Certificate Profile for Authorization. RFC 3281 Editor United States, 2002 - portal.acm.org

[38] Kobara K., Imai H. Pretty-Simple Password-Authenticated Key-Exchange under Standard Assumptions. IEICE Trans, pg 1-16, 2002

[39] Boyko V. All-or-Nothing Transforms and Password-Authenticated Key Exchange Protocols. PhD Thesis, MIT, Dept.of Electrical Engineering and Computer Science, Cambridge MA, pg 1-155, 2000

[40] Mackenzie P., Patel S., Swaminathan R. Password-Authenticated Key Exchange Based on RSA. Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology, Pages: 599 – 613, Vol. 1976, 2000

[41] http://en.literateprograms.org/Special:Downloadcode/Miller-Rabin_primality_test_%28Java%29

[42] Marco Casassa Mont. Public Key Infrastructure (X509 PKI). Trusted E-Services Laborator.HP Laboratories, Bristol, UK. 1994. Available at: http://www.google.co.uk/search?hl=en&safe=off&biw=1034&bih=605&noj=1&q=Public+Key+Infrastructure+Marco+Casassa+Mont&btnG=Search&aq=f&aqi=&aql=&oq=&gs_rfai=

[43] Dworkin M. Recommendation for Block Cipher Modes of Operation Methods and Techniques. Computer Security Division Information Technology Laboratory National Institute of Standards and Technology, pg 1-59, 2001

APPENDIX

Appendix A:

DECISIONAL DIFFIE-HELLMAN (DDH) ASSUMPTION:

Let $\mathcal{G}$ be a subgroup of $\mathbb{Z}_p^*$ of order q where p, q are prime, $q|p-1$, and $|q| = k$, the security parameter. Let g be a generator of $\mathcal{G}$. The DDH assumption states that it is infeasible for an adversary to distinguish between the following distributions:

$$\{x, y, z \leftarrow \mathbb{Z}_q : (g^x, g^y, g^{xz}, g^{yz})\} \text{ and } \{x, y, z, w \leftarrow \mathbb{Z}_q : (g^x, g^y, g^z, g^w)\}.$$

If the distribution is picked at random and an adversary A is given the element that is chosen at random, the adversary succeds if it gets the right distribution that was chosen; the advantage is defined as usual. Let $\varepsilon_{ddh}(k, t)$ be the maximum advantage the adversary has which runs in time t. The DDH assumption is that for $t = poly(k)$, the advantage $\varepsilon_{ddh}(k, t)$ is negligible.

ONE-TIME DIGITAL SIGNATURES:

Let $SigGen(1^k)$ be a probabilistic algorithm generating a public verification key/private signing key (VK,SK). Signing message M is denoted by $Sig \leftarrow Sign_{SK}(M)$, and verification is denoted by $b = Verify_{PK}(M, Sig)$. the signature is correct if $b = 1$. If adversary A is given (PK, M, Sig) and outputs a pair (M', Sig') which is not equal to the message/ signature given to it. The adversary's advantage is defined as the probability that $Verify_{PK}(M', Sig') = 1$. Let $\varepsilon_{sig}(k, t)$ be the maximum possible advantage of an adversary which runs in time t. The assumption is that for $t = poly(k)$, this value is negligible.

FORMAL DESCRIPTION OF THE PROTOCOL

Initialise (1^k) –
Select p, q prime with $|p| = k$ and $q|p-1$; this defines group $\mathcal{G}$
Choose random generators $g_1, g_2, h, c, d \leftarrow \mathcal{G}$
$\mathcal{H} \leftarrow$UOWHF
Publish parameters $(q, p, g_1, g_2, h, c, d, H)$
$< pw_c > C \in Client \leftarrow \{1, ..., N\}$

Fig.2 Specialisation of protocol initialisation [This was gotten from ref [3]]

Execute (Client, i, Server,j)-
(VK,SK) $\underleftarrow{R}$ $SigGen(1^k)$ $\quad x_1,x_2,y_1,y_2,z_1,z_2,w_1,w_2,r_1,r_2 \underleftarrow{R} \mathbb{Z}_q$
$A = g_1^{r_1}; B = g_2^{r_1}; C = h^{r_1} g_1^{pw_c}$ $\quad \alpha = \mathcal{H}(Client|VK|A|B|C)$
$D = (cd^{\alpha})^{r_1}$ $\quad msg - out_1 \leftarrow \langle Client|VK|A|B|C|D\rangle$
$E = g_1^{x_1} g_2^{x_1} h^{z_1} (cd^{\alpha})^{w_1}$ $\quad F = g_1^{r_2}; G = g_2^{r_2}; I = h^{r_2} g_1^{pw_c}$
$\beta = \mathcal{H}(Server|E|F|G|I)$
$J = (cd^{\beta})^{r_2}$ $\quad msg - out_2 \leftarrow \langle Server|E|F|G|I|J\rangle$
$K = g_1^{x_2} g_2^{y_2} h^{z_2} (cd^{\beta})^{w_2}$ $\quad msg - out_3 \leftarrow \langle K|Sign_{SK}(\beta|K)\rangle$
$sk_s^j \leftarrow sk_c^i \leftarrow A^{x_1} B^{y_1} (C.g_1^{-pw_c})^{z_1} D^{w_1} F^{x_2} G^{y_2} (I.g_1^{-pw_c})^{z_2} J^{w_2}$
$sid_s^j \leftarrow sid_c^i \leftarrow \langle msg - out_1|msg - out_2|msg - out_3\rangle$
return $\langle msg - out_1, msg - out_2, msg - out_3\rangle$

Reveal$(User, i)$-
return sk_u^i

Test$(User, i)$-
$b \underleftarrow{R} \{0,1\}$, $sk \leftarrow \mathcal{G}$
if $b = 0$ return sk else return sk_u^i

Fig.3. Specification of the Execute,Reveal, and Test oracles to which the adversary has access. Note that $q, g_1, g_2, h, c, d, \mathcal{H}$ are public, and $\mathcal{G}$ is the underlying group.Subscript S refers to the server, and C to the client.|This was gotten from ref [3]|

$\text{Send}_0(Client, i, Server)$-

$(VK, SK) \underleftarrow{R} SigGen\left(1^k\right) \qquad r \underleftarrow{R} \mathbb{Z}_q$

$A = g_1^r; B = g_2^r; C = h^r g_1^{pw_C} \qquad \alpha = \mathcal{H}(Client|VK|A|B|C)$

$msg - out \leftarrow \langle Client|VK|A|B|C|\left(cd^{\alpha}\right)^r\rangle$

$state_C^1 \leftarrow (SK, r, msg - out)$

return $msg - out$

$\text{Send}_1(Server, i, (Client, VK, A, B, C, D))$-

$x, y, z, w, r \underleftarrow{R} \mathbb{Z}_q \quad \alpha = \mathcal{H}(Client|VK|A|B|C) \quad E = g_1^x g_2^y h^z \left(cd^{\alpha}\right)^w$

$F = g_1^r; G = g_2^r; I = h^r g_1^{pw_C} \qquad \beta = \mathcal{H}(Server|E|F|G|I)$

$msg - out \leftarrow \langle Server|E|F|G|I|\left(cd^{\beta}\right)^r\rangle$

$state_s^1 \leftarrow (msg - in, x, y, z, w, r, \beta, msg - out)$

return $msg - out$

$\text{Send}_2(Client, i, \langle Server, E, F, G, I, J\rangle)$ –

$\langle SK, r, first - msg - out\rangle \leftarrow state_C^1 \quad \beta = \mathcal{H}(Server|E|F|G|I)$

$x, y, z, w \underleftarrow{R} \mathbb{Z}_q \qquad K = g_1^x g_2^y h^z \left(cd^{\beta}\right)^w$

$msg - out \leftarrow (K|sign_{SK}(\beta|K)) \; sid_C^1 \leftarrow (first - msg - out|msg - in|msg - out)$

$I^* = I.g_1^{-pw_C} \qquad sk_C^1 \leftarrow E^r F^x G^y \left(I^*\right)^z J^w$

return $msg - out$

$\text{Send}_3(Server, i, (K, sig))$ –

$\langle first - msg - in, x, y, x, w, r, \alpha, \beta, first - msg - out\rangle \leftarrow state_S^1$

$\langle VK, A, B, C, D\rangle \leftarrow first - msg - in$

$sid_S^1 \leftarrow \langle first - msg - in|first - msg - out|msg - in\rangle$

if $\text{Verify}_{VK}((\beta|K), Sig) = 1$

$C^* = C.g_1^{pw_C} \qquad sk_S^1 \leftarrow A^x B^y \left(C^*\right)^z D^w K^r$

else

$sk_S^1 \underleftarrow{R} \mathcal{G}$

return ε

Fig.4 Specification of the **Send** oracles to which the adversary has access. Note that $g, g_1, g_2, h, c, d, \mathcal{H}$ are public, and $\mathcal{G}$ is the underlying group. Subscript S refers to the server, and C to the client. The third argument to the **Send**oracles is denoted $msg - in$.|This was gotten from ref [3]|

Source Code:

```
package protocolInitialization;

import java.math.BigInteger;
import java.util.ArrayList;
import java.util.Random;
import org.apache.log4j.BasicConfigurator;

/**
 * This class belongs to the server
application; StockXchangeLatestPrice
 * This class executes the initialization
phase of the key exchange protocol as
 * specified in the protocol initialization
design on paper.
 * The final output is the public
information which is published in a
 * database as the most current public info
for computing the key exchange
 * protocol. Note that each time this class
is executed, the existing public info
 * is totally discarded, replacing it with
the new one generated.
 *
 * This class has a main method, which
allows the system Administrator to execute
 * this class as a unit application for the
purpose of generating a new public
 * info when ever there is a need for that.
 *
 * @author damola
 */
public class protocolIntialization {
    static org.apache.log4j.Logger logger =
org.apache.log4j.Logger.getLogger(protocolIn
tialization.class);
    private int p; //p is a prime and p = 2q
+ 1
    private int q;
    //To cope with the specification of the
development machine, we choose
    //prime number q between 1000 and 7000.
any thing above this may cause
    //the formGroup_G() method to execute
for ages (experiment reveals)
private static final int randomNumLimit =
7000;
    private static final int randomNumStart
= 1000;
    private ArrayList<BigInteger> group_G;
     /**
     * The constructor invokes all the
methods within itself. Hence all
computations
     * are done by instanciating this class
in the main method below
     * @param client
     */
    public protocolIntialization(){
        long startTm =
System.currentTimeMillis(); //start time for
experimental purpose

        BasicConfigurator.configure();
        logger.info("protocolIntialization
Starts....");
        group_G = new
ArrayList<BigInteger>();
        choose_p_q();
        formGroup_G();
        publishPublicInformation();

        long timeSpent =
(System.currentTimeMillis()-
startTm)/1000;//Time taken in seconds for
experimental purpose
        logger.info("protocolIntialization
Completed after.... "+ timeSpent +"
Seconds");
    }
    /**
     * This method computes the choosing of
p and q
     * q is choosen first at random such
that q is prime,(2q+1) is a prime
     * and q|(2q+1)(ie q is divisible by
(2q+1)).
     * When the above is all true, then
(2q+1) will become p ie p = (2q+1)
     * Note that a reliable algorithm
(MillerRabin32) is used for testing the
     * primality test of q and p.
     */
    public void choose_p_q(){
        logger.info("Start Choosing the
prime numbers p and q...");
        int x = 0, y = 0; //let q = x and p
= y
        boolean status = true;
        while (status){
            Random rand = new Random();
            x = rand.nextInt(randomNumLimit
- randomNumStart + 1) + randomNumStart;
            boolean isPrime =
MillerRabin32.miller_rabin_32(x);
            if (isPrime){
            //pick any prime number at
randome as q
            // x =  let the prime no. picked
be x
            //test for p ie if y = 2x + 1
                y = (2*x) + 1;
                isPrime =
MillerRabin32.miller_rabin_32(y);
                if(isPrime) status = false;
            //if y is prime then status =
false;
            //then control will leave the
while loop
            }
        }
        q = x;
        p = y;
       logger.info("p and q are Choosen such
that p = 2q+1... p = "+p+" q = "+q);
}
    /**
     * This method generates all the group
element from where the public info
     * g1, g2, h, c, d  are selected at
random. The value of the prime q chosen
     * determines the complexity of this
execution. Experiment revealed that chosing
     * a large prime would make this method
execute for hours because of the inability
     * of the development machine (laptop)
to cope with the large power computation
     * as well as looping for q number of
times
     */
    public void formGroup_G(){
        logger.info("Start Forming the Group
G...");
        int x = 0, i;
        BigInteger groupGenerator,
groupElement;
        //choose an element from Zp (ie
between 1 and 2q)
        Random rand = new Random();
```

```
        while (x == 0 || x == 1) x =
rand.nextInt((2*q)+1); // choose any element
of Zp^*(1, 2, ......., 2q)
        // x = number chosen
        groupGenerator =
BigInteger.valueOf(x);
        //squre x (to get the generator of
G)
        groupGenerator =
groupGenerator.pow(2); //square the element
choosen
         logger.info("Group generator chosen
and squared: "+groupGenerator);
        for(i = 0; i < q; i++){
            //do the modulus
            //ie x2 mod p = element of the
group G where x2 is a generator of G--an
element of Zp choosing at random and squred.
            //i is an integer between 1 and
x squred (x2)
            groupElement =
groupGenerator.pow(i);
            groupElement =
groupElement.mod(BigInteger.valueOf(p));
            group_G.add(groupElement);
        }
        logger.info("Finished Forming the
Group G. The members of the Group G are
below:\n"+group_G);
    }

    /**
     * This method selects from the group,
g1, g2, h, c, d and publish them in the
     * database as the current public info
for valid key exchange protocol execution
     */
    public void publishPublicInformation(){
        logger.info("Starts Selecting and
Publishing the public Info: p, q, g1, g2, h,
c, d ");
        int i, sizeOfG, index, numOfParam=5;
        int [] array = new int[numOfParam];
        sizeOfG = group_G.size();
        Random rand = new Random();

        for(i=0; i<numOfParam; i++){
            while ((index =
rand.nextInt(sizeOfG)) == 0){}

            array[i] =
group_G.get(index).intValue();
            System.out.println(array[i]+"
");
        }
        i = 0;
        DBmanager dbManager = new
DBmanager();
        dbManager.insertAllExceptHash(p, q,
array[i], array[i+1],array[i+2], array[i+3],
array[i+4]);
        logger.info("Finished Selecting and
Publishing the public Info on the
Database");
    }

    /**
     * main method to allow System
Administrator to run this class as a unit
     * application each time the company
needs to generate new and replace old
     * public information.
     * @param args
     */
    public static void main(String[] args) {
         new protocolIntialization();
     }
}//End of class protocolIntialization

//##### ANOTHER CLASS STARTS HERE ########
package passwordAuthenticationProtocol;

import Communication.CommunicationManager;
import java.io.UnsupportedEncodingException;
import java.math.BigInteger;
import
java.security.NoSuchAlgorithmException;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.apache.log4j.BasicConfigurator;

/**
 * This class is key in the key exchange
framework. ProtocolStep1 as the name implies
 * does all computation that is seen on the
top left hand side of the protocol design on
 * paper and is the step 1 of the protocol
execution. A collection of the result of the
computation forms the first message.
 * It does the computation and save each
result in the respective field and through
 * its Accessors methods (getters), allow
them to be retrieved when first message
 * is being collated. Note all methods here
are private except the Accessors method
 * this is for the security of this
application; to avoid any invocation of
these
 * expensive methods outside this class.
 * Note that the public info used in the
different methods here are first converted
 * from a primitive int to Big integer to
enable computation of large numbers, which
 * are imminent in security implimation like
this one
 *
 * @author damola
 */
public class ProtocolStep1 {
   static org.apache.log4j.Logger     logger
=
org.apache.log4j.Logger.getLogger(ProtocolSt
ep1.class);
    private int r1;
    private BigInteger A;
    private BigInteger B;
    private BigInteger C;
    private BigInteger D;
    private BigInteger alpha;
    private int VK=9;
    private int SK;
    private String clientName;
    private publicInformatnProvider
publicInfo = new publicInformatnProvider();
    private DBmanager db = new DBmanager();
    private Hashing hashing = new Hashing();

    /**
     * The constructor invokes all the
methods within itself. Hence all
computations
     * are done by instanciating this class
     * @param client
     */
    public ProtocolStep1(String client){
        BasicConfigurator.configure();
        clientName = client;
      logger.info("ProtocolStep1 Begins
execution..");
```

```
        Choose_r1();
        computeA();
        computeB();
        computeC();
        computeAlpha();
        computeD();

        logger.info("ProtocolStep1 Execution
Completed ....");
    }

    //methods
    /**
     * Method for choosing r1 from Zq
(0,....,q-1) at random
     */
    private void Choose_r1(){

        logger.info("Start Choosing r1 from
Zq (0 to "+(publicInfo.get_q()- 1)+")...");
        int x = 0;
        Random rand = new Random();
        while (x == 0 || x == 1) x =
rand.nextInt(publicInfo.get_q()- 1);
        r1 = x;
        //store r1 in the
CommunicationManager singleton object
        //because r1 will be reused in
Protocol Step3

CommunicationManager.getInstance().set_r1(x)
;
        logger.info("r1 is chosen! r1 = "+
r1);
    }

    /**
     * Method that computes A
     */
    private void computeA(){
        logger.info("ProtocolStep1 Start
computing A ...");
        BigInteger g1;
        g1 =
BigInteger.valueOf(publicInfo.get_g1());
        A =
(g1.pow(r1)).mod(BigInteger.valueOf(publicIn
fo.get_p()));
        logger.info("ProtocolStep1 Finished
computing A....  A = "+A);
    }

    /**
     * Method that computes B
     */
    private void computeB(){
        logger.info("ProtocolStep1 Start
computing B ...");
        BigInteger g2;
        g2 =
BigInteger.valueOf(publicInfo.get_g2());
        B =
(g2.pow(r1)).mod(BigInteger.valueOf(publicIn
fo.get_p()));
        logger.info("ProtocolStep1 Finished
computing B.... B = "+B);
    }

    /**
     * Method that computes C
     */
    private void computeC(){
       logger.info("ProtocolStep1 Start
computing C ...");
      BigInteger h =
BigInteger.valueOf(publicInfo.get_h());
      BigInteger g1 =
BigInteger.valueOf(publicInfo.get_g1());
      BigInteger p =
BigInteger.valueOf(publicInfo.get_p());
      int pw = db.fetchPassword();

      C =
((h.pow(r1)).mod(p)).multiply((g1.pow(pw)).m
od(p));
      C = C.mod(p);

     logger.info("ProtocolStep1 Finished
computing C.... C = "+C);
    }

    /**
     * Method that computes Alpha
     * alpha = H(clientName|VK|A|B|C|D)
     */
    private void computeAlpha(){
        logger.info("ProtocolStep1 Start
computing Alpha ...");

        //Achieve clientName|VK|A|B|C|D by
concatinating clientName, VK, A, B, C, D
       String concatVal = clientName +
Integer.toString(VK) + A.toString() +
B.toString() + C.toString();
        try {
            String hashValue =
hashing.hashValueOf(concatVal,
publicInfo.getHashAlgorithm());
            alpha =
hashing.intValueOfHash(hashValue);
          } catch
(UnsupportedEncodingException ex) {

Logger.getLogger(ProtocolStep1.class.getName
()).log(Level.SEVERE, null, ex);
        } catch (NoSuchAlgorithmException
ex) {
Logger.getLogger(ProtocolStep1.class.getName
()).log(Level.SEVERE, null, ex);
        }
       logger.info("ProtocolStep1 Finished
computing Alpha.... Alpha = "+alpha);
    }

    /**
     * Method that computes D
     */
    private void computeD(){
        logger.info("ProtocolStep1 Start
computing D ...");
        BigInteger d =
BigInteger.valueOf(publicInfo.get_d());
        BigInteger c =
BigInteger.valueOf(publicInfo.get_c());
        //alpha turns out to be vary large
and overwhelms the specification of the
laptop
        //(ie the development machine).
Hence alpha is divided by 10000000 every
where in the application
        //before being used.
        int a =
Math.abs(alpha.intValue())/10000000;
        System.out.println("a  is " +a);
        D = c.multiply(d.pow(a));
        D =
(D.pow(r1)).mod(BigInteger.valueOf(publicInf
o.get_p()));
```

```
        logger.info("ProtocolStep1 Finished
computing D....  D = "+D);
   }

 //Below are all the Accessor methods which
helps other classes to access the results
 //of the above computations from the
fields. They are all public
    public String getClientName(){
        return clientName;
    }
    public int getVeriKey(){
        return VK;
    }
    public BigInteger getA(){
        return A;
    }
    public BigInteger getB(){
        return B;
    }
    public BigInteger getC(){
        return C;
    }
    public BigInteger getD(){
        return D;
    }
}//End of class ProtocolStep1

####### ANOTHER CLASS STARTS HERE ######
package passwordAuthenticationProtocol;

import java.io.UnsupportedEncodingException;
import java.math.BigInteger;
import java.security.MessageDigest;
import
java.security.NoSuchAlgorithmException;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * This class belongs to the server
application; StockXchangeLatestPrice
 * This class has the responsiblity of
computing message digest
 * @author damola
 */
public class Hashing {

    /**
     * This method takes text and a name of
hash function and computes the message
     * digest of the text using the using
the hash function whose name is passed.
     * @param text
     * @param hashFunctionAlgorithm
     * @return message digest (msgDigest)
     * @throws UnsupportedEncodingException
     * @throws NoSuchAlgorithmException
     */
    public String hashValueOf(String
text,String hashFunctionAlgorithm) throws
UnsupportedEncodingException,
NoSuchAlgorithmException{
        byte [] digest, byteText;

        byteText = text.getBytes("UTF-8");
//convert the text to byte
        MessageDigest MD =
MessageDigest.getInstance(hashFunctionAlgori
thm);
        MD.update(byteText);
        digest = MD.digest();//compute the
digest
        //convert the digest from byte to a
hexadecimal string for consistency and
readability purpose
        StringBuffer hexString = new
StringBuffer();
              for (int
i=0;i<digest.length;i++) {
       hexString.append(String.format("%02x"
, digest[i]));
              }
                String msgDigest =
hexString.toString();
        return msgDigest;
    }

    /**
     * This method is dedicated to returning
a 128 bit value, which will later be used
     * else where in the application for AES
encryption. it takes a string of the session
     * key generated after the protocol
execution and pass it to an MD5 hash
function
     * to return a 128 bit value needed for
the encryption and decreption.
     * This method is very important because
it ensure that the key size for AES(128
bits)
     * is consistent since the session key
generated comes in different sizes.
     * @param text
     * @return 128 bits length digest
     */
    public byte [] MD5digestOf(String text){
         byte[] digest = null;
         byte[] byteText;
        try {
            byteText = text.getBytes("UTF-
8"); //convert text to byte
            MessageDigest MD =
MessageDigest.getInstance("MD5"); //get an
instance of messagedigest class with the
name of the algorithm
            MD.update(byteText);
            digest = MD.digest(); //compute
the digest
        } catch (NoSuchAlgorithmException
ex) {

Logger.getLogger(Hashing.class.getName()).lo
g(Level.SEVERE, null, ex);
        } catch
(UnsupportedEncodingException ex) {

Logger.getLogger(Hashing.class.getName()).lo
g(Level.SEVERE, null, ex);
        }
         return digest;
    }

    /**
     * This method takes the hex string from
hashValueOf method above and convert
     * it from hex to integer value (Big
integer). This method is kept separate since
     * the approach adopted in this
application is responsibility driven.
     * @param msg
     * @return intValue
     */
    public BigInteger intValueOfHash(String
msg){
        BigInteger intValue = new
BigInteger(msg, 16);
           return intValue;
```

```
    }
}//End of class Hashing
```

####### **ANOTHER CLASS STARTS HERE** ######

```
package passwordAuthenticationProtocol;

import java.io.UnsupportedEncodingException;
import
java.security.InvalidAlgorithmParameterExcep
tion;
import java.security.InvalidKeyException;
import
java.security.NoSuchAlgorithmException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import
javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
/**
 * This class has the responsibility of
carrying out symetric encryption and
decryption
 * using the 128 bits session key using the
AES algorithm with CBC mode and padding
scheme
 * to provide stronger encryption.
 * @author damola
 */
public class SessionKeyEncrypDecrypt {
    /**
     * This method takes the session key and
the data to be encrypted, encrypt
     * the data and return the ciphertext
     * @param sessionKey
     * @param data
     * @return cipherText
     * @throws
InvalidAlgorithmParameterException */
public byte [] Encrypt(byte [] sessionKey,
String data) throws
InvalidAlgorithmParameterException{
            byte[] cipherText = null,
message;
      try {
    message = data.getBytes("UTF-8");
            //use the 128bits session key to
create a SecretKeySpec object which can
            //used in java for encryption
            SecretKey key128bits = new
SecretKeySpec(sessionKey, "AES");
            //use AES with CBC mode &
Padding for stronger encryption
            Cipher c =
Cipher.getInstance("AES/CBC/PKCS5Padding");
            IvParameterSpec ivspec = new
IvParameterSpec(sessionKey);
            c.init(Cipher.ENCRYPT_MODE,
key128bits, ivspec);
            //do the encryption
            cipherText = c.doFinal(message);
        }
        catch (IllegalBlockSizeException ex)
{

Logger.getLogger(SessionKeyEncrypDecrypt.cla
ss.getName()).log(Level.SEVERE, null, ex);
        } catch (BadPaddingException ex) {
Logger.getLogger(SessionKeyEncrypDecrypt.cla
ss.getName()).log(Level.SEVERE, null, ex);
        }        catch (InvalidKeyException
ex) {
Logger.getLogger(SessionKeyEncrypDecrypt.cla
ss.getName()).log(Level.SEVERE, null, ex);
        }        catch
(NoSuchAlgorithmException ex) {
Logger.getLogger(SessionKeyEncrypDecrypt.cla
ss.getName()).log(Level.SEVERE, null, ex);
        } catch (NoSuchPaddingException ex)
{
Logger.getLogger(SessionKeyEncrypDecrypt.cla
ss.getName()).log(Level.SEVERE, null, ex);
        }        catch
(UnsupportedEncodingException ex) {

Logger.getLogger(SessionKeyEncrypDecrypt.cla
ss.getName()).log(Level.SEVERE, null, ex);
        }
        return cipherText;
}
    /**
     * This method takes the session key and
the ciphertext to be decrypted, decrypt
     * the cipher and return the message
     * @param sessionKey
     * @param cipher
     * @return  message (plaintext)
     * @throws
InvalidAlgorithmParameterException
     */
    public String Decrypt(byte []
sessionKey, byte [] cipher) throws
InvalidAlgorithmParameterException{
            byte[] message=null;
      try {
           //use the 128bits session key to
create a SecretKeySpec object which can
            //used in java for decryption
            SecretKey key128bits = new
SecretKeySpec(sessionKey, "AES");
            Cipher c =
Cipher.getInstance("AES/CBC/PKCS5Padding");
            IvParameterSpec ivspec = new
IvParameterSpec(sessionKey);
            c.init(Cipher.DECRYPT_MODE,
key128bits, ivspec);
            //do the decryption
     message = c.doFinal(cipher);
        }
        catch (IllegalBlockSizeException ex)
{ other exception caught and handle here
Pleas see program file submitted}
        return (new String(message));
    }
}//End of class SessionKeyEncrypDecrypt
```