Two Secure Implementations of Sockets and Their Tradeoffs

Ahren Studer
Computer Science Department
University of Rochester
astuder@cs.rochester.edu

Abstract

As electronic communication grows in popularity and replaces paper as an accepted official medium, the need for secure electronic communication increases. Using typical socket classes, data is sent over the network in a format so that anyone who reads the data will be able to interpret it correctly. Encryption is only part of the solution to this problem. The other part is getting the key needed for encryption or decryption from the client to the server or vice versa. Public key encryption does offer a solution to this problem, but the computationally intensive nature does not provide acceptable throughput. In order to achieve acceptable throughput and still be secure certain protocols can be used to manage a session key between two computers without compromising security. This paper covers the implementation of two such protocols, the tradeoffs made for ease of programming and use, and the performance of each protocol on a real network. The final results find that the computation needed for public key encryption is approximately one-tenth of a second slower than trusted third party on current technology. However, a load under 100 clients could cause the trusted arbitrator's queue for service to grow so large that public key encryption could perform faster.

1 Introduction

With the expansion of computers and electronic communication into almost every facet of human interaction, there is often a need to limit access to information. When data is stored on an object and physically sent to the recipient, it is simple to prevent access by limiting physical access. However, when data is transmitted over a wire, or across several networks, multiple parties may be able to view that same information by simply noting the signals sent. The solution to this problem is to encrypt the data is some manner, turning it into meaningless bits. However, encryption itself is not such a simple answer since the two parties must obtain keys in a manner so that eavesdroppers will not also have the keys. The work of this paper is to generate a set of classes for use in Java to get the key used for communication to both end users, without anyone else accessing it, and once the key is established making sure the cipher used can not be easily broken.

The remainder of the paper is broken into 5 sections: some background about ciphers and key management, the different protocols this work uses to ensure only the end users have access to the data, the tradeoffs made for ease of use rather than perfect security, a look at the performance of the two methods, and a final section with some concluding remarks.

2 Background

Several different ciphers have been used over the centuries to encrypt data, ranging from the simple shift cipher to state of the art symmetric ciphers like the current Advanced Encryption Standard (AES), Rijndael [1] or asymmetric ciphers such as ElGamal [3]. With symmetric key ciphers, knowledge of the encryption key means the decryption key is easily acquired. With asymmetric ciphers, the same is not true, but the computations required relatively takes much more time than any symmetric ciphers.

Some ciphers can easily be cracked using pattern matching, statistics or other techniques, providing the key or the meaning of the ciphertext within a short amount of time, usually a few hours. How-

ever, most asymmetric ciphers, when properly implemented, and some symmetric ciphers like AES and it's predecessor DES have only been decrypted using brute force attacks. During these attacks every possible key is tried until a logical answer is generated from the ciphertext. Obviously this takes a large amount of time. However, with a key space of 58 bits, DES is no longer considered secure due to the increasing speed of computers. For this reason Rijndael was developed with a variable key size of 128, 192, or 256 bits, allowing for much greater security, even from brute force attacks. With AES's increased security, relatively fast processing, in comparison to asymmetric ciphers, and acceptance as the cipher system of choice for vital documents by the NIST, it seems to be the logical choice for the basis of secure electronic communication. The problem of getting the key for AES to both users without any unwanted parties accessing it is the focus of the protocols discussed in the next section.

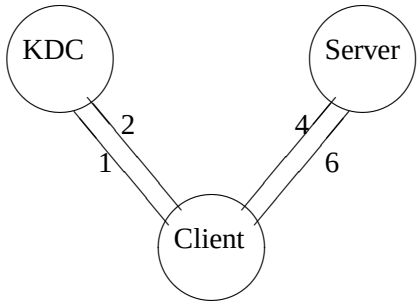
3 Key Exchange Protocols

In order for symmetric encryption systems to be useful the key must be sent to both parties in some manner that prevents any unwanted party from receiving the data. To achieve this goal the following two methods are used. The first relies on the security of the RSA cipher [4], and is open to a few attacks. The second depends on a trusted third party similar to a slimmed down Kerberos [5]. This is not as vulnerable to attacks, but the centralization of crucial information has the typical drawbacks of limited scalability and a single point of failure.

3.1 RSA Key Exchange

The key exchange problem was one of the original reasons for the push for public key cryptography. With separate keys for encryption and decryption it is very simple to allow for anyone to encrypt a message, here the key, which only the owner of the private key can decrypt. With RSA, there are only a few steps to get the session key from the client to the server. Once a normal connection is made between the two computers, the server sends its public key to the client. Now the client encrypts the session key using the server's public key, and sends the ciphertext back. At this point the client has a local copy of the session key, and the only version of the session key that is transmitted can only be decrypted by the server's pubic key. Once the server gets the encrypted version of the session key it uses its private key to decrypt the message, and both machines have a copy of the session key.

3.2 Trusted Key Distribution Center (KDC)



KDC protocol Messages

Using a trusted Key Distribution Center (KDC) and a few messages a session key can easily be transferred between the client and the server. The previous figure shows the steps needed for key management in this protocol. It is also crucial to note that due to object oriented programming and that the KDCSocket and KDCServerSocket classes are children of the Socket class the first real connection is made between the client and the server. With this traditional tcp connection established the following steps are performed to get a secure session key.
(1) The client sends the KDC its IP address and the IP address of the server it wishes to connect to.
(2) The KDC responds with a ticket $\{K_{sess}, S_{addr}, C_{addr}, TimeStamp, LifeTime, \{T\}K_s\}K_c$. In this ticket K = key, S = server, C = client, T is a copy of ticket, and anything within brackets is encrypted using the key to the right of the bracket (i.e. the T ticket is encrypted with the server's key and the clients key).
(3)Now the client decrypts the ticket acquiring the session key along with the other info.
(4) The client forwards $\{T\}K_s$ to the server.
(5)The server decrypts the ticket.
(6) If the client is truly who the ticket says it is the server responds with $\{Timestamp + 1\}K_{sess}$, signifying it really is the server and it has received

the session key.

4 Tradeoffs

In order for these protocols to appear transparent to the final user, and for ease of programming when using the developed classes, several issues had to be addressed. For each of the implementations different issues had to be addressed.

One common issue that needed addressed for both of the classes was private key management. Since both classes should appear to the final user as though no password is needed, the password must be stored somewhere locally. For security to be maintained it would be ideal that users could not access these files, and then simply decrypt any future messages being sent to that machine. However, the file that stores the passwords must be readable by the program executing on part of the users. For this reason the classes should be implemented as a system call so the users do not have to know machine's passwords to use the KDC or the private key for the RSA implementation, and malicious users will not be able to access the password files. Currently with the classes implemented in Java this issue is yet to be resolved.

4.1 RSA tradeoffs

The current weakness of this key exchange protocol is its vulnerability to a man in the middle attack. If the class attempts to make a connection to a server and another hosts intercepts the request it can respond with its own public key, establish a connection to the client, and generate its own connection to the original desired server and simply forward messages between the two while being able to view all of the data. The use of a certificate authority (CA) could be used to prevent this. With a CA in place, the server would send back its public key encrypted by the CA's private key. In this scenario, the client would then decrypt the message using the CA's public key and acquire the server's public key. However, with the class implemented at the user level it would be simple for anyone to alter the program so that a different CA could be used, replacing the true CA's public key with a key defined by the attacker.

To face this problem this work decided to limit the transparency of the underlying class. Whenever a connection is made to the server, the client checks a cache of previously acquired ip address/public key pairs. If the server responds with a different public key than the one stored in the cache, the user is notified and given the choice to proceed or not along with a display of the new public key. The same options and information are presented to the user when a connection is made to a server whose ip address is not in the cache.

4.2 KDC tradeoffs

The major weakness, and strength, of this key exchange protocol is the centralization of vital information. The KDC must have knowledge of every ip address/private key pair of every system that may use this KDC. Despite the advantage of guaranteed authentication of clients and users, this method also results in poor scalability, overhead of extra connections to the KDC, and a possible focus of attack.

With all key negotiations being routed through the KDC as the number of users increases the latency of generating a connection will also increase. This problem could be addressed by distributing the private key list to multiple machines and have copies of one KDC running on several different machines. This solution does raise other problems though. An obvious issue with many copies of the ip addr/private key list is race conditions when a key is changed at one location and other KDC's are relying on stale values. Another problem with distributing the multiple copies of the KDC is how to control what copy a program connects to. A central server that redirects request may actually increase time versus simply fulfilling requests due to the added I/O of another connection. Letting the programmer decide the address of the KDC to be used could be an alternative, but leads to greater trouble. Instead the simple solution of a single KDC was used.

The added overhead of connecting with the KDC before being able to securely connect with the end server added a surprising amount of overhead when compared to the time required to connect and receive authentication from the server. This vast difference is partially the result of having to wait for a new thread to handle the request. However, the majority of the time is a result of file access that is delayed until the KDC is contacted. In order to save time this file access is done as soon as a connection is made to

the server, allowing for it to occur in parallel with the client's communication with the KDC. The exact values for the time it takes to get a ticket from the KDC and the authentication response from the server is discussed in section5.

The centralization of all the private keys to one machine is a risk. If the file at the KDC is ever accessed the attacker has the private key for every machine that could generate connections through that KDC. Here the weakness of the KDC being a user level process is most evident. With the KDC running as a user program, its private key file is also accessible by the user. To address this a system administrator could start the program, allowing only privileged key management programs such as add user or change password to access the password file. Again by treating passwords on a per ip address basis, users are freed from certain responsibilities, but security can be easily compromised if the proper precautions are not taken.

5 Performance Comparison

To analyze the performance of the two key exchange protocols measurements were made to find on average how long it took to make a connection to the end server and the different operations involved with this action. Measurements were taken about the delay that results from AES encryption and decrpytion. However, according to the measurment techniques available in Java, the time required to operate on a message was 0ms. Previous work by Caltagirone and Anantha found that AES with 128-bit block size can achieve throughput ranging from 230 MBps to 427 MBps depending level of parrallelization and implementation method, hardware vs. software [2].

5.1 RSA Protocol Performance

Attribute	Client Machine	Server Machine
Processor	Pent. III	Pent. IV w/ HT
Proc. Speed	930 MHz	3.2 GHz
Local Net	100 Mbps	100 Mbps

Table 1: Machine Specs for RSA Evaluation

To evaluate the performance of the RSA key exchange protocol two hundred connections were made from one client to a server on another machine across campus and the time it took to connect was measured using Java's `System.currentTimeMillis()`. During testing no other programs were running on either machine that would consume processing power or network bandwidth. Table 1 shows some values for the machines used during testing. The timing results are shown in table 2.

5.2 KDC Protocol Performance

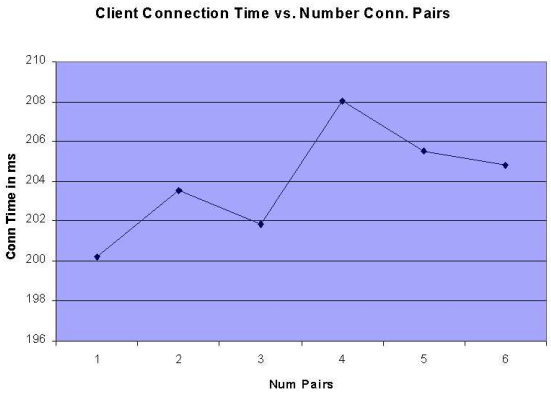


Figure 1: Connection times for the KDC protocol

The data in figure 1 was collected using the same type of machines used for the RSA performance data collection. However, every client/server pair used the 930 MHz computer, and the KDC was the 3.2 GHz hyper-threaded machine. Table 2 gives a breakdown of the time needed to make connections when only one connection is being made to the KDC.

Operation	Average Time Taken[1]
Typical TCP Connection	1ms
RSA Connection	303 ms
RSA Encryption	820 μs
RSA Decryption	91.8 ms
Response from KDC	199.2 ms
Authentication from Server	2.1 ms

Table 2: Timing Results for Varying Operations of Importance

[1] Java's timing methods only allow for measurement of time in

6 Performance Analysis

From the data collected, not much can be said about the two protocols that is not already known. The RSA protocol took approximately one-tenth of a second longer to establish a connection than the KDC protocol. Also with the limited number of machines used for evaluation of the KDC protocol it would be difficult to extrapolate the point where the computation required by RSA would be faster than the network communication.

As expected the RSA protocol did not perform as well as the KDC protocol due to its computational intensity. However, as shown in table 2 the amount of computation needed is asymmetric, leaving the majority of the work on the decryption end. This is the result of the mathematics behind RSA. Typically a small exponent is used for the public key, in this case three, requiring a small number of multiplications. However, for decryption the exponent, d, is the multiplicative inverse of the encryption exponent mod φ(N) [4], requiring roughly ln(d) multiplications, which is a large number for an N of 1024 bits. It is also important to note that the time accounted for in the table only adds up to 93 ms. The remainder of this time is consumed by network travel time and more importantly file I/O to retrieve the keys at the server side, and check the cache on the client side.

The data in table 2 and figure 1 show that even with a minor load the KDC protocol achieves much better performance than the RSA protocol. Examining the values one thing that seems unusual is the long time it takes for the KDC to return a ticket. This is a result of the file I/O (note 200 ms the same amount of time for file I/O for the RSA protocol) at the KDC to find the client's and the server's private keys. Once the ticket is returned the remainder of the protocol is relatively fast and is simply a result of network communication, decryption, computation, encryption and sending the authentication information.

The limited amount of data available determines a limitation for predicting when the RSA protocol is faster than the KDC protocol. However, a trend line can be placed on the data in figure 1 to generate the equation $ConnTime = 200.49e^{0.0049n}$. Setting this equation to the average time to establish a connection using the RSA protocol one finds that it would require more than 84 clients requesting a service at the same moment for the slowest one to achieve performance worse than the RSA protocol.

millisecond increments. For this reason the timing of a tcp connection and RSA encryption are questionable, and the operation may consume more or less than the time quoted in the table.

7 Concluding Remarks

This work has developed two nearly transparent Java classes that offer secure communication when properly implemented, and analyzed the latency of making a connection with these two classes on a real network.

Both of these classes require almost no user interaction for normal operation, and only minimal work for the programmer to change any TCP socket program into a TCP socket program with AES encryption and secure key exchange. The RSA key exchange method allows for two programs to communicate without the involvement of a third trusted party. However, it is vulnerable to a man in the middle attack if the user does not know in advance what the server's public key is. The KDC key exchange method is not vulnerable to a man in the middle attack, but suffers from several other weaknesses. The major one is the requirement that both machines involved must register their IP address/private key pair with the KDC. There is also a single point of failure, meaning if the KDC loses power or is corrupted secure communication is no longer possible.

As for the performance of the two classes the results were as expected. Using a large portion of the machines available up to twelve computers were used at a time to collect timing results. Even though the load on the KDC was increasing, on average the response time did not seem strongly related to the load. However, the important fact that the KDC protocol is faster than the RSA protocol is evident by the difference of 100 ms between the two protocols' latencies. With the limited data, a trend line was extrapolated that predicts the KDC will outperform the RSA protocol as long as less than 84 clients are requesting tickets at a time.

Despite its inherent need for more work on the administrative level the KDC protocol does benefit from faster connection time, complete transparency to the end user, and lack of vulnerabilities. However, in a system where hundreds of requests may be made

simultaneously, the use of the RSA protocol may perform better due to the lack of a centralized service.

This work has shown the advantages and disadvantages of two protocols that, when properly implemented, can provide secure communication. It also demonstrates that in order for security the underlying method often is not completely transparent to the user, requiring more time to establish a connection or even verifi cation of crucial values, like the public key in the RSA scheme when no information is in the cache.

References

[1] Announcing the advanced encryption standard (aes). *Federal Information Processing Standards*, Publication 197, 2001.

[2] C. Caltagirone and K. Anantha. High throughput, parallelized 128-bit aes encryption in a resource-limited fpga. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 240–241. ACM Press, 2003.

[3] T. ElGamal. A public key cryptosystem and signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31:469–473, 1985.

[4] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM*, 21:120–126, 1978.

[5] J. Steiner, C. Neuman, and J. Schiller. Kerberos: An authentication server for open network systems. In *the USENIX Winter Conference*, pages 191–202, 1988.