Implementing a Distributed Key Algorithm to Enhance Confidentiality and Provide Fault Tolerance

Isaac Rego
Information Systems Department
St. Cloud State University
Saint Cloud, MN 56301
reis0601@stcloudstate.edu

Nicole Gillespie
English Department
St. Cloud State University
Saint Cloud, MN 56301
E-mail: gini0604@stcloudstate.edu

Dennis Guster
IS Department
St. Cloud State University
Saint Cloud, MN 56301
dcguster@stcloudstate.edu

Abstract

This paper attempts to present an implementation method for splitting encryption keys among multiple hosts. Thereby, protecting its identity if any given host within the host group was compromised at the root level. Further, the importance of providing fault tolerance within the host group is recognized and a fault tolerance related solution is proposed.

1 Introduction

While few would argue that today's widely used encryption algorithms lack robustness, they are readily available. What makes each session that uses such algorithms unique is the key which can be simplistically viewed as a seed number. Therefore, gaining access to the key provides access to the data or the reverse is also true; lose the key and access can be denied to legitimate users. It is then crucial to not only keep the key disclosed, but also ensure it does not get lost or destroyed as well. A classic scenario might be to store the key (using some type of access protection) within a regular file system on a Unix host. If hackers obtain root access to that host, then the key(s) are potentially compromised. To combat this problem the authors have implemented an algorithm that stores portions of the key across several hosts and provides fault tolerance in the event of host failure. The algorithm is designed to allow the number of hosts used to be varied. For example, a simple scenario might involve 3 hosts (N) with a (N-1) logic, whereby if any one host fails the key can still be determined. Further, the number of pieces the key is split among and the order those pieces are stored and transmitted can also be varied.

Further, with the implementation of a distributed key server, an adversary would need to gain access to many hosts to obtain the full key, versus just one in a traditional model. For example, in an N-3 model using 10 hosts, the hacker would need to take control of 7 hosts to get all the necessary key fragments. In a case in which a hacker was able to penetrate just 6 hosts, the fragments would not provide enough information to reassemble the key. If the placement of the physical hosts is well thought out (for example not all logical zones in the same physical host) then breaking into that many host while not impossible will be difficult and time consuming. The algorithm was implemented in a UNIX script file and safe guards such as md5sum are used to help ensure that the files containing the key (or portions thereof) have not been tampered with.

2 Background

Object Oriented Programming emphasizes the reuse of code [18]. By adopting this logic, the sophistication and robustness of the majority of encryption algorithms currently in use is quite impressive [1]. Because people are sharing the same algorithms, it is critical that each person's encryption session is different. To do this, many of these techniques when deployed in an operational setting make use of keys to ensure that each encryption session is unique (or at least very close to unique). The most common example of this methodology is secure shell (ssh), in which every client connecting to a given ssh daemon uses the same encryption algorithm, but is assigned different keys. In effect, that key becomes a "seed" number so each client's encrypted message would look different.

Therefore, a vulnerability in this methodology is present no matter how robust the algorithm whenever the key is compromised. Also of concern is that the widely used algorithms are well published. There is a wealth of information on the Internet that directly delineates the formulas they use, see for example [17].
Vulnerabilities related to the present strategies have to some degree been acknowledged, and the need for improved encryption strategies recognized. A good example of this situation is described via the testimony to the US Senate by Barbara Simons of the Association of Computer Machinery (ACM). She testified that the current capability of hackers to obtain their own clandestine computing clusters results in encrypted data that is especially vulnerable to brute force attacks. Such testimony in turn supports the need for sound key management [13]. Effective key management appears in the literature as an important topic for discussion, see for example [4]. Further, a sound historical starting point to review is a book chapter edited by Filipe and Obaidat [6], and for those interested in the ramifications of wireless transmission an appropriate example is provided by work [10].

A vital principle of sound key deployment is effectively controlling how the key is generated and keeping it disclosed once it is generated. A common means used to implement those principles is through key agreement protocols, in which information from both sides of a client/server model is used to generate the shared key [6]. In this model, the hacker has to compromise two completely different hosts to derive the key. Increasing the number of hosts that need to be compromised certainly makes it more difficult for the hacker and logically, would appear to be a well-founded methodology. Building on that basic premise, deriving a key from multiple sources can then be easily expanded. In literature within the field, there are examples of such strategies. For example, a key translation center can be used to impart a third entity in order to provide the appropriate evaluation of the encrypted method [3]. Further, Sun Micro-Systems [14] presents a number of methods that could be deployed to obtain efficient data encryption in the enterprise, but suggests the key, and its management, is the primary foundation for those methods.

It is clear that the importance of the key cannot be overlooked when two very important facts are evaluated: first, if the key is compromised, sensitive data can be read by the "bad guys;" second, if the key is lost, the data cannot be read by the "good guys" [19]. In other words, protecting the key from outside intruders and loss prevention by users are critical. For example, a hacker compromises a host, obtains access to the key, modifies the key in some way, and the auto-security-scripts catch the hacker's attack. The hacker is then banned from the enterprise on the network firewall level, however, bad things can still happen. While the first thought is that some sensitive data could be compromised, the

hacker might modify or destroy the key as well. In this case, if a redundant key strategy is not in place, then data may be lost.
It is clear that some type of redundancy is required, but increasing the number of key instances raises the probability that the encrypted data can be compromised. More specifically, a tradeoff exists between redundancy level and potential key vulnerability. Others see the key as one of the most crucial components when devising an encryption strategy. In [11], the author suggests that key length should be adequately assessed so that sufficient robustness is achieved and claims that encryption keys and passwords should be stored in escrow with a secure third party. This strategy has merit in the sense that the third party concept outsources the responsibility of key security and places the key on a host external to where the data is stored. However, the key might still be lost if the third parties' secure host is compromised, and that host if provided by an organization such as Verisign might be a tempting target in its own right to a hacker. Nonetheless, it is certainly clear that it is important to establish an effective key management plan. One could successfully argue that key management is a prime consideration in the productive use of encryption and research that addresses the vulnerabilities of key storage generation. However, effectiveness is still needed, especially in regard to key storage. The literature points out that key management is especially critical in large security groups when a member leaves the group [15]. As we continue to become more mobile, the issue of key generation and management also becomes a crucial issue in a wireless world. This is especially problematic in satellite applications, in which the key generator may need to be on board the satellite. Thus, a reliable means is needed to ensure the keys are protected and synchronized with ground stations [5].

3 Logic for Splitting the Key

When implementing a plan involving splitting elements of the key, basic logic is simply to limit access to the entire key. Given today's enterprise-oriented world in which autonomous systems feature multiple hosts in a distributed environment, the task is easy to accomplish, because pieces of the key can be stored across several hosts. This results in a strategy where if any given host is compromised then only a portion of the key will be compromised. Of course without the entire key, any single portion has limited value. To describe the strategy in a general sense when fault tolerance is implemented, a couple of basic examples will be depicted below. In the first basic algorithm, which is not fault tolerant, key positions are simply alternated across the key servers. To illustrate this basic premise, an example will be delineated involving four server hosts with a key size of twelve characters and a server key size of 3 characters. A key position table is displayed below as Table 1:

svr	n=4		key=12 svrKey=3
a	1	5	9
b	2	6	10
c	3	7	11
d	4	8	12

Table 1: Key Position Distribution for 4 Servers and Key Size = 12.

Table 1 illustrates that the distributed server host architecture will consist of four servers, a-d, each server will host 3 characters (svrKey) of the total 12 character key. Server a will host relative key positions 1,5,9, and server b will host key positions 2,6,10, and so forth. In other words, each character alternates in sequence across all 4 server hosts. A major limitation of this design is that the pattern is predictable and fault tolerance is not included. With this design, when any of the 4 hosts fails, then the encryption/de-encryption process is compromised. If added robustness is desired, the number of server hosts used and the key size can be varied to obtain different levels of complexity. Just like in other forms of encryption, the complexity of the design needs to be balanced because added complexity can result in less reliability and performance degradations. Therefore, to provide fault tolerance, what would be required in a basic design? The second basic example described below provides fault tolerance by distributing the key across four different hosts using a key length of sixteen bytes. Once again, the goal in this example is to ensure that the key never entirely resides on one host. As a matter of fact, one might expect reliability problems with any one host resulting in a need to provide fault tolerance for this important security process. The simple solution offered is an example that invokes N-2 reliability logic, which means that the key could still be extracted if any two of the four hosts fail. In this design, the key structure would appear as follows (where the numbers represent relative byte position in the key):

Host 1: 1 2 3 4 5 6 7 8 9 10 11 12
Host 2: 5 6 7 8 9 10 11 12 13 14 15 16
Host 3: 9 10 11 12 13 14 15 16 1 2 3 4
Host 4: 13 14 15 16 1 2 3 4 5 6 7 8

In this example, each host has only ¾ of the key (12 out of the 16 bytes), but the key can be built from any two of the hosts. For example, when host number three fails, the key can still be obtained by getting positions 1-12 from host 1 and 13-16 from host2. In this example, host 4 could be non-operational as well. Given a second scenario which features the failure of both host 1 and 2, the key could then be derived from host 3 and 4 with host 3 providing key positions: 1-4; 9-12 and host 4 providing positions 5-8. While the example could be greatly improved from a security perspective, however, it does

illustrate the basic premise that if a host is compromised, the entire key is not. In other words, this design might be looked at as an example of augmented layering distribution. That is essentially what is occurring between the server key segments, whereby each segment is slightly overlapping the other [19].

The implementation example depicted herein in section 3 uses a basic key splitting algorithm. This algorithm takes an RSA key, divides the number of lines and provides storage for them on multiple hosts. To make this relationship work, there are a minimum number of bytes for any key, and that number is a function of the number of key servers (hosts) selected. This topic is covered in [b], and an updated summary of this relationship appears below.

Where:
s = Key Server to pull key from
skp = server key position on server s
i = interval (or offset)
kp = key position on key being built
SvrKeyLen = Server Key Length
KeyLen = Key Length

Given 4 servers and an offset of 2 then i = 2
KeyLen = 12
SvrKeyLen = 8

The formulae are:

skp = kp - (s - 1) • i
    for all skp <= SvrKeyLen

skp = kp + KeyLen - ((s - 1) • i)
    for all skp <= KeyLen

The Basic Code would be:

```
if kp - ((s - 1) * i) <= SvrKeyLen then
    Skp = kp - ((s - 1) * i)
elseIf kp + KeyLen - ((s - 1) * i) <= KeyLen then
    skp = kp + KeyLen - ((s - 1) * i)
else
    skp = 0 'or ""
```

end if

The remainder of this paper deals with the implementation of such an algorithm on a practical level. Specifically, a UNIX script has been devised that splits an RSA—which stands for Rivest, Shamir and Adleman, the people that first publicly described it—and stores pieces of that key across several different hosts. More specifically, a pre-generated key is read in as a file, and the line-by-line and output are treated as fractions of the file in the working directory. The fragments are immediately joined, and the MD5sum hash is checked to verify file integrity.

Next, a list of the files is created because the number of entries could be dynamic as determined by the size of the key. Then, this output list is passed to a temporary file, which in our script is named nxqlist.txt. Figure 1 below depicts the computing architecture required to support such a key management scheme. This architecture is based on standard routing and firewalling techniques, but feature a key controller and multiple authentication/key server from which the original key is generated so that the stored key never resides entirely on any single host.

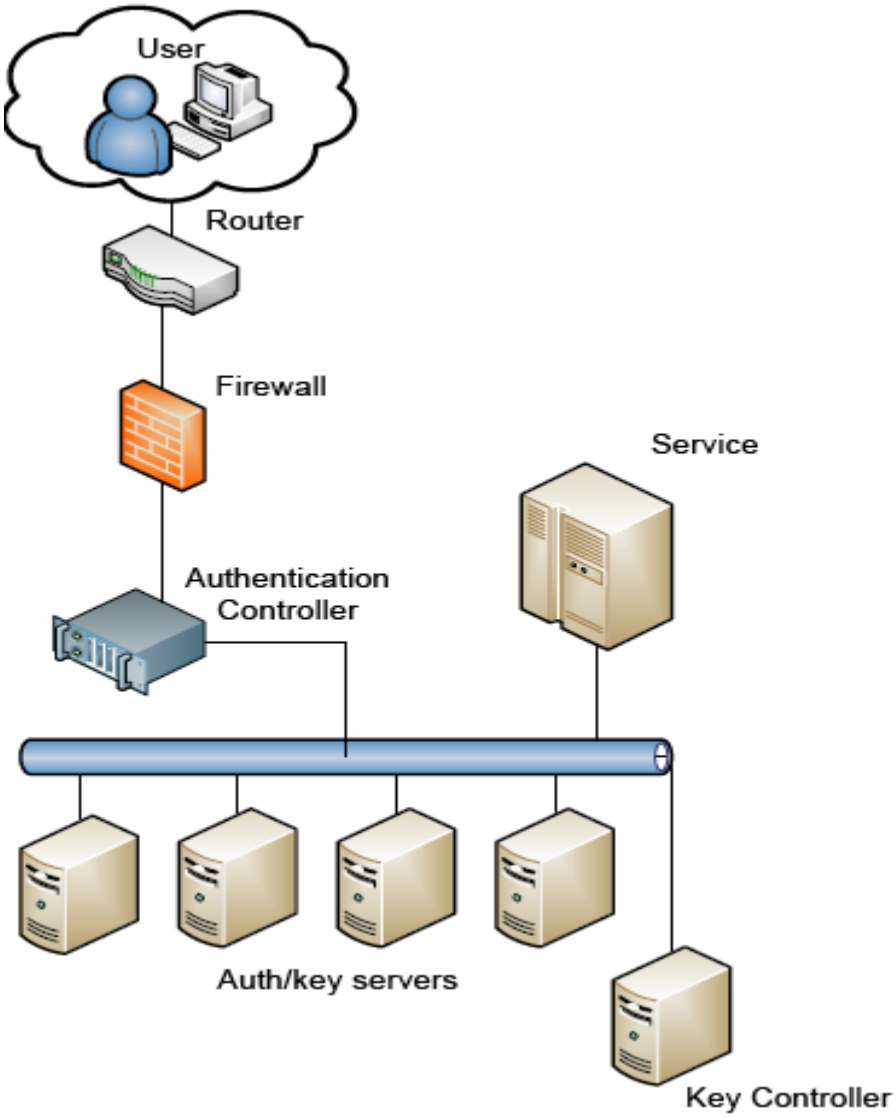

Figure 1: Distributed Key Architecture

4 Implementation of the Code

The actual UNIX script code is included in Appendix A. The implementation of this code results in the configuration of a file named hostlist.txt and assumes the existence of the RSA key. The hostlist.txt file is intended to be a configuration file that is used to determine the destination of the key fragments. The number of entries in the file can be dynamic. Typically, the fragments will be sent to as many hosts that are indicated in the file.

Example entries:
irego@127.0.0.1:~/folder1/
irego@127.0.0.1:~/folder2/
irego@127.0.0.1:~/folder3/

The script begins by reading in the RSA key file line-by-line to the end of the file. Reading in line-by-line is necessary only for proof of concept, which is the main goal at this stage. Each line is output into its own file with the prefix "nxq" to define a unique string from which to identify it later. The "nxq" string is appended by accounting for the suffix size beginning at 100 to the count of lines to the end of the file. At the end of each line in each file, the bar symbol "|" is used as a delimiter. The original key's MD5sum follows this, which is included to assure the integrity of the key when the fragments are joined back together.

After the key is fragmented and the pieces are outputted to files, they are immediately joined again using "cat" and placed into a temporary file. An MD5sum is then calculated and compared to the MD5sum of the original RSA key. If these two hashes match, the script continues on. This is a protection mechanism to eliminate errors that may prevent all of the pieces from being properly extracted from the original key.
While searching for the unique string "nxq1," a file list is generated, as all the fragments will begin with this filename. A file list is compiled and stored in a temporary file named "nxqlist.txt". The hostlist.txt file is then read line by line and placed into an array, with each line being a record. At the same time, the file list nxqlist.txt is read line by line and each line placed as a record into an array. Both arrays are output in a fashion that allows the location of each file to be paired with a destination specified in hostlist.txt. This is all output forwarded to a file named "sendlist.txt".

Example entry in send list:
nxq101.txt irego@127.0.0.1:~/folder1/

The sendlist.txt file is then read line by line, taking each line as a literal command string. The UNIX secures copy command "scp" is used to ensure security within the host. Ssh is used to ensure each fragment of the key can then be securely transferred across the network to the destinations specified in hostlist.txt.

The fragments can be rebuilt by using ssh to login to each key/auth server, searching the directories specified by hostlist.txt for the "nxq" prefix, then requesting them and using "cat" to join them back together. This can all be done without creating files on the hard disk.

A verification method must be used to compare each MD5sum located in each fragment after the "|" delimiter. Using a method of excluding any MD5s's that simply don't match the rest could be useful. However, a better method is to keep the original MD5sum on the key controller and compare the MD5s in each fragment to that value. In this method, there could be considerable delays in authentication when there are intensive request patterns. This is also a function of the design in regard to the number of fragments utilized. In planning for such an implementation, it is important not to rely on performance averages, rather a look at outliers would be wiser. Planning for the known maximum workload could help mitigate a potentially serious denial of service vulnerability. Further, placing all of this information on a single host, if only temporarily, still creates a short-term vulnerability. Consequently, this scenario needs to be evaluated carefully because if all the fragments are compromised and the MD5sums are all changed, there would be no way of telling if the key has maintained its integrity.

5 Summary

The prime purpose of this paper is to devise a prototype of a distributed authentication system that provides redundancy and fault tolerance against both nature disasters and intrusion. The final implementation will be composed of five parts on each key server as indicated in the diagram shown above (Figure 1). Again, these parts include:

A service in listening state on a socket that is used for connecting to the key controller.
A service in listening state waiting inbound authentication requests.
A service waiting for connections from other key servers.
An available socket used to connect to each key server from other key servers.
An outbound socket connection to the authentication controller, which will also keep track of load levels.

Once the services above are active, an inbound request comes in and is sent to the authentication controller that determines which key/auth server should handle the request, based on the most recent data the key servers sent. The key/auth server then queries the other key/auth servers to gather all of the fragments of the key to authorize access to this user. This key/auth server now determines if the fragments have maintained their integrity. Once the key has been read, the fragments are immediately purged from the memory.

6 Discussion:

While the examples presented herein lack the complexity needed for a production system, it is hoped that the basic logic for such a system has been demonstrated. Obviously the key complexity is a function of the encryption algorithm used and the number of segments it is broken into. However, the logic is such that the number of fragments could be easily expanded. Still, there may be a trade in regard to complexity versus performance. Deriving the key is also a critical portion of the algorithm. The basic method used herein is to simply generate it on a "master" host and then split and store appropriately thereafter. This method is vulnerable because the whole key resides on a single host for a short period of time. A second method would fit well with the four-server model presented in Figure 1 of this paper. In this model, sever 1 could generate positions (or lines) 1-6, sever 2 positions 7-12, and then the resulting values could be distributed accordingly. In this case at least the whole key is never entirely on one host. There are certainly other and more complex variants of this basic premise. In regard to situations when the physical host is compromised that could allow access to the whole key. Further, if all fragments of the key are comprised, and the hacker is able to alter the MD5sums to the same signature, integrity cannot be known. Another thing to consider is if a significant number of fragments of the key are known, then brute force could be used on the remaining fractions of the key resulting in ascertaining its entirety.

Also, the manner in which the key or "key portions" are generated is cause for concern. The first method one would consider is to use some type of classical random number generator, and if sophisticated methodology is employed, this method may in-fact be adequate. However, given the work of Shannon [12], it is clear that there are limitations even with the best classical system. Therefore, one might consider generating the random numbers using some type of quantum system. Devices are now available from $1,500 to $3,000, and marketed from companies such as http://www.idquantique.com/index.php/component/content/article/6.html.

7 Conclusion

This paper attempts to present an implementation method for splitting the encryption key among multiple hosts so that the whole key would never totally reside on one host. Thereby, protecting its identity if any given host within the host group was compromised at the root level. Further, the importance of providing fault tolerance within the host group was recognized and a fault tolerance related solution was proposed. While this paper lays some ground-work for implementing such a solution, much work and refinement is needed before this solution would be appropriate for deployment in a production environment.

References

[1] Abdalla, M., Bellare, M. and Neven, G. (2008). A Provable-Security Treatment of Robust Encryption. Cryptology ePrint Archive, Report 2008/440.

[2] Ateniese, G., Fu, K., Green, M. and Hohenberger. (2006). Improved Proxy Re-encryption Schemes with Applications to Secure Distributed Storage. ACM Transactions on Information and Systems Security. 9:1, 1-30.

[3] Boyd, C. and Mathuria, A. Protocols for Authentication and Key Establishment. Springer, Berlin. 10 .

[4] Cavaiani, C. and Alves-Foss, J. (1996). A Mutual Authentication Protocol with Key Distribution in a Client/Server Environment. http://www.acm.org/crossroads/xrds2-4/authen.html.

[5] European Space Agency. (2006). Industrial Policy Committee: List of Intended Invitations to Tender. ESA/IPC(2006)11.Rev.11.

[6] Filipe, J. and Obaidat, M. (2008). Two Types of Key-Compromise Impersonation Attacks against One-Pass Key Establishment Protocols. In: Communications in Computer and Information Science, 23, 227-238.

[7] Halderman , J. et al. (2009). Lest we Remember: Cold-boot Attacks on Encryption Keys. Communications of the ACM. 52:5.

[8] Hong-Wei, Z., Shi-Hui, P. and Ping-Ping, L. (2007). Distributed Key Establishment Scheme in Wireless Sensor Networks. Eight ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, Qingdao, China, 2: 268-273.

[9] JHU-MIT Proxy Re-cryptography Library. (2007). http://spar.isi.jhu.edu/prl/.

[10] Li, J. and Garuba, M. (2008). Encryption as an Effective Tool in Reducing Wireless LAN Vulnerabilities. Proceedings of the Fifth International Conference on Information Technology: New Generations, 557-562.

[11] Moore, F. (2005). Preparing for Encryption: New threats, Legal Requirements Boost Need for Encrypted Data. Computer Technology Review, August-September.

[12] Shannon, Claude (1949). "Communication Theory of Secrecy Systems". Bell System Technical Journal 28 (4): 656–715.

[13] Simons, B. (1996).Testimony to the US Senate Regarding S.1726, the “Promotion of Commerce On-line in the Digital Era.

[14] Sun MicroSystems. (2007). Encryption Strategies: The Key to Controlling Data, Sun Services White Paper, August, http://www.sun.com/storage/docs/Encryption_Strategies_wp.pdf.

[15] Tseng, Y. (2003). A Scalable Key Management Scheme with Minimizing Key Storage for Secure Group Communications. International Journal of Network Management. 13:6, 419-425.

[16] US patent 6026163, http://www.freepatentsonline.com/6026163.html.

[17] Yang, H. (2007). Cryptography Tutorials: Herong's Tutorial Notes, http://www.herongyang.com/crypto/.
[18] Basili, V., Briand, L. and Melo, W. (1996). How Reuse Influences Productivity in Object-oriented Systems. Communications of the ACM. 39(10).
[19] Guster, Dennis, Brown, Christopher, Sultanov, Renat (2010). Enhancing Key Confidentiality by Distributing Portions of the Key Across Multiple Hosts, Midwest Instructional Computing Symposium Annual Meeting in Eau Claire, WI, April 16-17, 2010.

Appendix A: Unix Scripts

```
#! /bin/bash
#Isaac Rego 2011 - Business Computer Research Laboratory, St. Cloud State University, MN
#made using GNU bash, version 3.2.48(1)-release (i486-pc-linux-gnu)
#Pre-requisites, check bash version, RSA key in working dir, hostlist.txt in working dir
bash version.bash #runs the version check
#**************CLEANUP***********
cleanup(){
	rm nxq1*
	rm sendlist.txt
}
#***************JOIN*****************
join(){ #joins all split the files
	cat nxq1* | cut -d'|' -f1 > nxqfile
}
#****************COMPARE****************
comparemd5(){
	rm md5*
	nxqMD5=`md5sum nxqfile | cut -d' ' -f1`
	echo "$nxqMD5"
	echo "$orgMD5"
	if [ "$nxqMD5" == "$orgMD5" ]; then
		echo "md5's match"; else
		echo "Md5's do NOT match, or error occurred"; exit 0
	fi
#source:http://stackoverflow.com/questions/239526/truncate-output-in-bash
}
#***************GROUP******************
group(){
```

```
        ls | grep nxq1 > nxqlist.txt
        #declare the array
        declare -a arrayOut
        # Force reading of each line into one element of the array.
        line_cntOut=0
        while :
        do
                read data_lineOut
                if [ $? -ne 0 ]; then
                        break
                fi
                arrayOut[$line_cntOut]="$data_lineOut"
                (( line_cntOut++ ))
           done  <  nxqlist.txt
#declare the array.
declare -a arrayIn
# Force reading of each line into one element of the array.
line_cntIn=0
while :
do
        read data_lineIn
        if [ $? -ne 0 ]; then
                break
        fi
        arrayIn[$line_cntIn]="$data_lineIn"
        (( line_cntIn++ ))
done  <  ~/PROJECT/hostlist.txt
line_numIn=0; line_numOut=0
while [ $line_numOut -lt $line_cntOut ]; do
        echo "${arrayOut[$line_numOut]}
${arrayIn[$line_numIn]}${arrayOut[$line_numOut]}" >> sendlist.txt
        (( line_numIn++ )); (( line_numOut++ ))
        if [ $line_numIn = $line_cntIn ]; then
                line_numIn=0
        fi
done
#source http://www.linuxquestions.org/questions/linux-newbie-8/bash-reading-file-into-
array-848005/
}
#******************SEND*******************
```

```
send(){
while read sendline; do
	scp $sendline
done < sendlist.txt
}
#***************SPLIT/MAIN*********************
#echo "Enter num hosts: "
#read nHosts
key=~/PROJECT/rsakey
count=100 #because starting at 0 or 1 doesn’t work for that 10 is before 1 in the file list so catting doesn’t work
keyLen=`wc -l $key | cut -d' ' -f1`
orgMD5=`md5sum $key | cut -d' ' -f1`
while read line; do
	(( count++ ))
	echo "$line|$orgMD5" > nxq$count.txt #unique string for starting file name so only those unique ones are deleted
done < $key
#***********RUN FUNCTIONS***********
join #runs join function
comparemd5 # runs comparemd5 function
group #runs group function
send #runs send function
cleanup #deletes all files this made
```